**Kafka**

## 一、Kafka概论

**1为什么需要消息系统**

（1）.解耦：

　　允许你独立的扩展或修改两边的处理过程，只要确保它们遵守同样的接口约束。

（2）.冗余：

　　消息队列把数据进行持久化直到它们已经被完全处理，通过这一方式规避了数据丢失风险。许多消息队列所采用的"插入-获取-删除"范式中，在把一个消息从队列中删除之前，需要你的处理系统明确的指出该消息已经被处理完毕，从而确保你的数据被安全的保存直到你使用完毕。

（3）.扩展性：

　　因为消息队列解耦了你的处理过程，所以增大消息入队和处理的频率是很容易的，只要另外增加处理过程即可。

（4）.灵活性 & 峰值处理能力：

　　在访问量剧增的情况下，应用仍然需要继续发挥作用，但是这样的突发流量并不常见。如果为以能处理这类峰值访问为标准来投入资源随时待命无疑是巨大的浪费。使用消息队列能够使关键组件顶住突发的访问压力，而不会因为突发的超负荷的请求而完全崩溃。

（5）.可恢复性：

　　系统的一部分组件失效时，不会影响到整个系统。消息队列降低了进程间的耦合度，所以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。

（6）.顺序保证：

　　在大多使用场景下，数据处理的顺序都很重要。大部分消息队列本来就是排序的，并且能保证数据会按照特定的顺序来处理。（Kafka 保证一个 Partition 内的消息的有序性）

（7）.缓冲：

　　有助于控制和优化数据流经过系统的速度，解决生产消息和消费消息的处理速度不一致的情况。

（8）.异步通信：

　　很多时候，用户不想也不需要立即处理消息。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，但并不立即处理它。想向队列中放入多少消息就放多少，然后在需要的时候再去处理它们。

**2 Kafka是一种分布式的，基于发布/订阅的消息系统。**

**Kafka是最初由Linkedin公司开发，是一个分布式、支持分区的（partition）、多副本的（replica），基于zookeeper协调的分布式消息系统，它的最大的特性就是可以实时的处理大量数据以满足各种需求场景：比如基于hadoop的批处理系统、低延迟的实时系统、storm/Spark流式处理引擎，web/nginx日志、访问日志，消息服务等等，用scala语言编写，Linkedin于2010年贡献给了Apache基金会并成为顶级开源 项目。**

**特点：**

- 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。

- 可扩展性：kafka集群支持热扩展

- 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

- 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）

- 高并发：支持**数千个**客户端同时读写

**3 常用Message Queue对比**

RabbitMQ

RabbitMQ是使用Erlang编写的一个开源的消息队列，本身支持很多的协议：AMQP，XMPP, SMTP, STOMP，也正因如此，它非常重量级，更适合于企业级的开发。同时实现了Broker构架，这意味着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都有很好的支持。

Redis

Redis是一个基于Key-Value对的NoSQL数据库，开发维护很活跃。虽然它是一个Key-Value数据库存储系统，但它本身支持MQ功能，所以完全可以当做一个轻量级的队列服务来使用。对于RabbitMQ和Redis的入队和出队操作，各执行100万次，每10万次记录一次执行时间。测试数据分为128Bytes、512Bytes、1K和10K四个不同大小的数据。实验表明：入队时，当数据比较小时Redis的性能要高于RabbitMQ，而如果数据大小超过了10K，Redis则慢的无法忍受；出队时，无论数据大小，Redis都表现出非常好的性能，而RabbitMQ的出队性能则远低于Redis。

ZeroMQ

ZeroMQ号称最快的消息队列系统，尤其针对大吞吐量的需求场景。ZMQ能够实现RabbitMQ不擅长的高级/复杂的队列，但是开发人员需要自己组合多种技术框架，技术上的复杂度是对这MQ能够应用成功的挑战。ZeroMQ具有一个独特的非中间件的模式，你不需要安装和运行一个消息服务器或中间件，因为你的应用程序将扮演了这个服务角色。你只需要简单的引用ZeroMQ程序库，可以使用NuGet安装，然后你就可以愉快的在应用程序之间发送消息了。但是ZeroMQ仅提供非持久性的队列，也就是说如果宕机，数据将会丢失。其中，Twitter的Storm 0.9.0以前的版本中默认使用ZeroMQ作为数据流的传输（Storm从0.9版本开始同时支持ZeroMQ和Netty作为传输模块）。

ActiveMQ

ActiveMQ是Apache下的一个子项目。 类似于ZeroMQ，它能够以代理人和点对点的技术实现队列。同时类似于RabbitMQ，它少量代码就可以高效地实现高级应用场景。

Kafka/Jafka

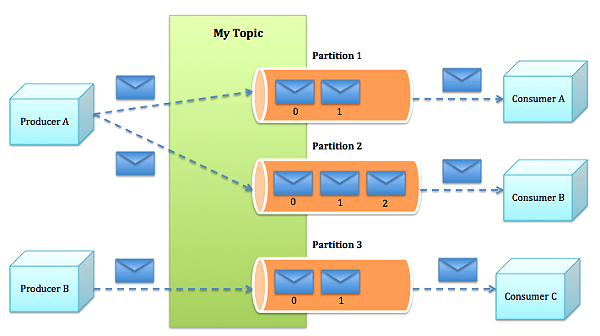
Kafka是Apache下的一个子项目，是一个高性能跨语言分布式发布/订阅消息队列系统，而Jafka是在Kafka之上孵化而来的，即Kafka的一个升级版。具有以下特性：快速持久化，可以在O(1)的系统开销下进行消息持久化；高吞吐，在一台普通的服务器上既可以达到10W/s的吞吐速率；完全的分布式系统，Broker、Producer、Consumer都原生自动支持分布式，自动实现负载均衡；支持Hadoop数据并行加载，对于像Hadoop的一样的日志数据和离线分析系统，但又要求实时处理的限制，这是一个可行的解决方案。Kafka通过Hadoop的并行加载机制来统一了在线和离线的消息处理。Apache Kafka相对于ActiveMQ是一个非常轻量级的消息系统，除了性能非常好之外，还是一个工作良好的分布式系统。

**Kafka的吞吐量高达17.3w/s,不愧是高吞吐量消息中间件的行业老大。这主要取决于它的队列模式保证了写磁盘的过程是线性IO。此时broker磁盘IO已达瓶颈**

**RocketMQ也表现不俗，吞吐量在11.6w/s磁盘IO %util已接近100%。**

**RabbitMQ的吞吐量5.95w/s，CPU资源消耗较高。它支持AMQP协议，实现非常重量级，为了保证消息的可靠性在吞吐量上做了取舍。我们还做了RabbitMQ在消息持久化场景下的性能测试，吞吐量在2.6w/s左右。**

# 4 因为Kafka内在就是分布式的，一个Kafka集群通常包括多个代理。为了均衡负载，将话题分成多个分区，每个代理存储一或多个分区。多个生产者和消费者能够同时生产和获取消息。

****

过程：

1.Producer根据指定的partition方法（round-robin、hash等），将消息发布到指定topic的partition里面

2.kafka集群接收到Producer发过来的消息后，将其持久化到硬盘，并保留消息指定时长（可配置），而不关注消息是否被消费。

3.Consumer从kafka集群pull数据，并控制获取消息的offset

原理：

生产者使用自己的序列化方法对消息内容进行编码。然后向broker发起消息。为了提高效率，一个发布请求中可以包含一组消息。

消费者订阅话题，并为话题创建一个或多个消息流。发布到该话题的消息被均衡的分发到这些流中。

每个消息流为不断产生的消息提供了迭代接口。

消费者迭代流中每一条消息，并处理消息的有效负载。

迭代器不会停止。如果当前没有消息，迭代器将阻塞直至有新的消息发布到该话题

**5 zookeeper在kafka中的作用**

1)Broker注册

Broker在zookeeper中保存为一个临时节点，节点的路径是/brokers/ids/[brokerid],每个节点会保存对应broker的IP以及端口等信息.

2)Topic注册

在kafka中,一个topic会被分成多个区并被分到多个broker上，分区的信息以及broker的分布情况都保存在zookeeper中，根节点路径为/brokers/topics,每个topic都会在topics下建立独立的子节点，每个topic节点下都会包含分区以及broker的对应信息，例如下图中的状态

3)生产者负载均衡

当Broker启动时，会注册该Broker的信息，以及可订阅的topic信息。生产者通过注册在Broker以及Topic上的watcher动态的感知Broker以及Topic的分区情况，从而将Topic的分区动态的分配到broker上.

4)消费者

kafka有消费者分组的概念，每个分组中可以包含多个消费者，每条消息只会发给分组中的一个消费者，且每个分组之间是相互独立互不影响的。

5)消费者与分区的对应关系

对于每个消费者分组，kafka都会为其分配一个全局唯一的Group ID,分组内的所有消费者会共享该ID,kafka还会为每个消费者分配一个consumer ID,通常采用hostname:uuid的形式。在kafka的设计中规定，对于topic的每个分区，最多只能被一个消费者进行消费，也就是消费者与分区的关系是一对多的关系。消费者与分区的关系也被存储在zookeeper中节点的路劲为 /consumers/[group\_id]/owners/[topic]/[broker\_id-partition\_id],该节点的内容就是消费者的Consumer ID

6)消费者负载均衡

消费者服务启动时，会创建一个属于消费者节点的临时节点，节点的路径为 /consumers/[group\_id]/ids/[consumer\_id],该节点的内容是该消费者订阅的Topic信息。每个消费者会对/consumers/[group\_id]/ids节点注册Watcher监听器，一旦消费者的数量增加或减少就会触发消费者的负载均衡。消费者还会对/brokers/ids/[brokerid]节点进行监听，如果发现服务器的Broker服务器列表发生变化，也会进行消费者的负载均衡

7)消费者的offset

在kafka的消费者API分为两种(1)High Level Api：由zookeeper维护消费者的offset (2) Low Level API,自己的代码实现对offset的维护。由于自己维护offset往往比较复杂，所以多数情况下都是使用High Level的APIoffset在zookeeper中的节点路径为/consumers/[group\_id]/offsets/[topic]/[broker\_id-part\_id],该节点的值就是对应的offset

8）需要要消费者知道现在那些生产者（对于消费者而言，kafka就是生产者）是可用的。

如果没了zookeeper消费者如何知道呢？

如果每次消费者在消费之前都去尝试连接生产者测试下是否连接成功，效率呢？

所以kafka需要zk，在kafka的设计中就依赖了zk了。

**6 一些概念**

**- Kakfa Broker Leader的选举**

Kakfa Broker集群受Zookeeper管理。所有的Kafka Broker节点一起去Zookeeper上注册一个临时节点，因为只有一个Kafka Broker会注册成功，其他的都会失败，所以这个成功在Zookeeper上注册临时节点的这个Kafka Broker会成为Kafka Broker Controller，其他的Kafka broker叫Kafka Broker follower。。这个Controller会监听其他的Kafka Broker的所有信息。

如果这个kafka broker controller宕机了，在zookeeper上面的那个临时节点就会消失，此时所有的kafka broker又会一起去Zookeeper上注册一个临时节点，因为只有一个Kafka Broker会注册成功，其他的都会失败，所以这个成功在Zookeeper上注册临时节点的这个Kafka Broker会成为Kafka Broker Controller，其他的Kafka broker叫Kafka Broker follower。

例如：一旦有一个broker宕机了，这个kafka broker controller会读取该宕机broker上所有的partition在zookeeper上的状态，并选取ISR列表中的一个replica作为partition leader（如果ISR列表中的replica全挂，选一个幸存的replica作为leader;如果该partition的所有的replica都宕机了，则将新的leader设置为-1，等待恢复，等待ISR中的任一个Replica“活”过来，并且选它作为Leader；或选择第一个“活”过来的Replica（不一定是ISR中的）作为Leader），这个broker宕机的事情，kafka controller也会通知zookeeper，zookeeper就会通知其他的kafka broker。

这里曾经发生过一个bug，TalkingData使用Kafka0.8.1的时候，kafka controller在Zookeeper上注册成功后，它和Zookeeper通信的timeout时间是6s，也就是如果kafka controller如果有6s中没有和Zookeeper做心跳，那么Zookeeper就认为这个kafka controller已经死了，就会在Zookeeper上把这个临时节点删掉，那么其他Kafka就会认

为controller已经没了，就会再次抢着注册临时节点，注册成功的那个kafka broker成为controller，然后，之前的那个kafka controller就需要各种shutdown去关闭各

种节点和事件的监听。但是当kafka的读写流量都非常巨大的时候，TalkingData的一个bug是，由于网络等原因，kafka controller和Zookeeper有6s中没有通信，于是

重新选举出了一个新的kafka controller，但是原来的controller在shut down的时候总是不成功，这个时候producer进来的message由于Kafka集群中存在两个kafka controller

而无法落地。导致数据淤积。

**- consumer group消费**

当启动一个consumer group去消费一个topic的时候，无论topic里面有多个少个partition，无论我们consumer group里面配置了多少个consumer thread，这个consumer group下面

的所有consumer thread一定会消费全部的partition；即便这个consumer group下只有一个consumer thread，那么这个consumer thread也会去消费所有的partition。因此，最优

的设计就是，consumer group下的consumer thread的数量等于partition数量，这样效率是最高的。

同一partition的一条message只能被同一个Consumer Group内的一个Consumer消费。不能够一个consumer group的多个consumer同时消费一个partition。多个Consumer Group下的consumer可以消费同一条message.

一般来说

（1）一个Topic的Partition数量大于等于Broker的数量，可以提高吞吐率。

（2）同一个Partition的Replica尽量分散到不同的机器，高可用。

当add a new partition的时候，partition里面的message不会重新进行分配，原来的partition里面的message数据不会变，新加的这个partition刚开始是空的，随后进入这个topic的message就会重新

参与所有partition的load balance

**- Partition Replica**

每个partition可以在其他的kafka broker节点上存副本，以便某个kafka broker节点宕机不会影响这个kafka集群。存replica副本的方式是按照kafka broker的顺序存。

例如有5个kafka broker节点，某个topic有3个partition，每个partition存2个副本，那么partition1存broker1,broker2，partition2存broker2,broker3。。。以此类推

（replica副本数目不能大于kafkabroker节点的数目，否则报错。这里的replica数其实就是partition的副本总数，其中包括一个leader，其他的就是copy副本）。

如果某个broker宕机，其实整个kafka内数据依然是完整的。但是，replica副本数越高，系统虽然越稳定，但是回来带资源和性能上的下降；replica副本少的话，也会造成系统丢数据的风险。

（1）怎样传送消息：producer先把message发送到partition leader，再由leader发送给其他partition follower。

（2）在向Producer发送ACK响应消息前需要保证有多少个Replica已经收到该消息：根据ack配的个数而定

（3）怎样处理某个Replica不工作的情况：如果这个部工作的partition replica不在ack列表中，就是producer在发送消息到partition leader上，partition leader向partition follower发送message没有响应而已，这个不会影响整个系统，也不会有什么问题。

（4）怎样处理Failed Replica恢复回来的情况：如果这个partition replica之前不在ack列表中，那么启动后重新受Zookeeper管理即可，之后producer发送message的时候，partition leader会继续发送message到这个partition follower上。如果这个partition replica之前在ack列表中，此时重启后，需要把这个partition replica再手动加到ack列表中。（ack列表是手动添加的，出现某个部工作的partition replica的时候自动从ack列表中移除的）

**- Partition leader与follower**

partition也有leader和follower之分。leader是主partition，producer写kafka的时候先写partition leader，再由partition leader push给其他的partition follower。

partition leader与follower的信息受Zookeeper控制，一旦partition leader所在的broker节点宕机，zookeeper会冲其他的broker的partition follower上选择follower变为parition leader。

**- Topic分配partition和partition replica的算法：**

（1）将Broker（size=n）和待分配的Partition排序。

（2）将第i个Partition分配到第（i%n）个Broker上。

（3）将第i个Partition的第j个Replica分配到第（(i + j) % n）个Broker上

**- 消息投递可靠性三种模式**

第一种是啥都不管，发送出去就当作成功，这种情况当然不能保证消息成功投递到broker；

第二种是Master-Slave模型，只有当Master和所有Slave都接收到消息时，才算投递成功，这种模型提供了最高的投递可靠性，但是损伤了性能；

第三种模型，即只要Master确认收到消息就算投递成功；实际使用时，根据应用特性选择，绝大多数情况下都会中和可靠性和性能选择第三种模型

**- 消息在broker上的可靠性**

因为消息会持久化到磁盘上，所以如果正常stop一个broker，其上的数据不会丢失；但是如果不正常stop，可能会使存在页面缓存来不及写入磁盘的消息丢失，这可以通过配置flush页面缓存的周期、阈值缓解，但是同样会频繁的写磁盘会影响性能，又是一个选择题，根据实际情况配置。

**- message状态**

在Kafka中，消息的状态被保存在consumer中，broker不会关心哪个消息被消费了被谁消费了，只记录一个offset值（指向partition中下一个要被消费的消息位置），

这就意味着如果consumer处理不好的话，broker上的一个消息可能会被消费多次。

**-message持久化**

Kafka中会把消息持久化到本地文件系统中，并且保持o(1)极高的效率。我们众所周知IO读取是非常耗资源的性能也是最慢的，这就是为了数据库的瓶颈经常在IO上，需要换SSD硬盘的原因。但是Kafka作为吞吐量极高的MQ，却可以非常高效的message持久化到文件。这是因为Kafka是顺序写入o（1）的时间复杂度，速度非常快。也是高吞吐量的原因。

由于message的写入持久化是顺序写入的，因此message在被消费的时候也是按顺序被消费的，保证partition的message是顺序消费的。一般的机器,单机每秒100k条数据。

**- Produer**

Producer向Topic发送message，不需要指定partition，直接发送就好了。kafka通过partition ack来控制是否发送成功并把信息返回给producer，producer可以有任意多的thread，这些kafka服务器端是不care的。

Producer端的delivery guarantee默认是At least once的。也可以设置Producer异步发送实现At most once。Producer可以用主键幂等性实现Exactly once

**- Kafka高吞吐量**

Kafka的高吞吐量体现在读写上，分布式并发的读和写都非常快，写的性能体现在以o(1)的时间复杂度进行顺序写入。读的性能体现在以o(1)的时间复杂度进行顺序读取，

对topic进行partition分区，consume group中的consume线程可以以很高能性能进行顺序读。

**- Kafka delivery guarantee(message传送保证)**

（1）At most once消息可能会丢，绝对不会重复传输；

（2）At least once 消息绝对不会丢，但是可能会重复传输；

（3）Exactly once每条信息肯定会被传输一次且仅传输一次，这是用户想要的。

**- 批量发送**

Kafka支持以消息集合为单位进行批量发送，以提高push效率。

**- push-and-pull**

Kafka中的Producer和consumer采用的是push-and-pull模式，即Producer只管向broker push消息，consumer只管从broker pull消息，两者对消息的生产和消费是异步的。

**- Kafka集群中broker之间的关系**

不是主从关系，各个broker在集群中地位一样，我们可以随意的增加或删除任何一个broker节点

**- 负载均衡方面**

Kafka提供了一个 metadata API来管理broker之间的负载（对Kafka0.8.x而言，对于0.7.x主要靠zookeeper来实现负载均衡）

**- 同步异步**

Producer采用异步push方式，极大提高Kafka系统的吞吐率（可以通过参数控制是采用同步还是异步方式）。

**- 分区机制partition**

Kafka的broker端支持消息分区partition，Producer可以决定把消息发到哪个partition，在一个partition 中message的顺序就是Producer发送消息的顺序，一个topic中可以有多个partition，具体partition的数量是可配置的。

partition的概念使得kafka作为MQ可以横向扩展，吞吐量巨大。partition可以设置replica副本，replica副本存在不同的kafka broker节点上，第一个partition是leader,其他的是follower，message先写到partition leader上，再由partition leader push到parition follower上。所以说kafka可以水平扩展，也就是扩展partition。

**- 离线数据装载**

Kafka由于对可拓展的数据持久化的支持，它也非常适合向Hadoop或者数据仓库中进行数据装载。

**- 实时数据与离线数据**

kafka既支持离线数据也支持实时数据，因为kafka的message持久化到文件，并可以设置有效期，因此可以把kafka作为一个高效的存储来使用，可以作为离线数据供后面的分析。当然作为分布式实时消息系统，大多数情况下还是用于实时的数据处理的，但是当cosumer消费能力下降的时候可以通过message的持久化在淤积数据在kafka。

**- 基本名词**

Broker：Kafka节点，一个Kafka节点就是一个broker，多个broker可以组成一个Kafka集群。

Topic：一类消息，消息存放的目录即主题，例如page view日志、click日志等都可以以topic的形式存在，Kafka集群能够同时负责多个topic的分发。

Partition：topic物理上的分组，一个topic可以分为多个partition，每个partition是一个有序的队列

Segment：partition物理上由多个segment组成，每个Segment存着message信息

Producer : 生产message发送到topic

Consumer : 订阅topic消费message, consumer作为一个线程来消费

Consumer Group：一个Consumer Group包含多个consumer, 这个是预先在配置文件中配置好的。各个consumer（consumer 线程）可以组成一个组（Consumer group ），

partition中的每个message只能被组（Consumer group ） 中的一个consumer（consumer 线程 ）消费，如果一个message可以被多个consumer（consumer 线程 ） 消费的话，那么这些consumer必须在不同的组。Kafka不支持一个partition中的message由两个或两个以上的consumer thread来处理，即便是来自不同的consumer group的也不行。它不能像AMQ那样可以多个BET作为consumer去处理message，这是因为多个BET去消费一个Queue中的数据的时候，由于要保证不能多个线程拿同一条message，所以就需要行级别悲观所（for update）,这就导致了consume的性能下降，吞吐量不够。而kafka为了保证吞吐量，只允许一个consumer线程去访问一个partition。如果觉得效率不高的时候，可以加partition的数量来横向扩展，那么再加新的consumer thread去消费。这样没有锁竞争，充分发挥了横向的扩展性，吞吐量极高。这也就形成了分布式消费的概念。

**- 持久化**

kafka使用文件存储消息(append only log),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是非常艰难的.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.对于kafka而言,较高性能的磁盘,将会带来更加直接的性能提升.

**- 性能**

除磁盘IO之外,我们还需要考虑网络IO,这直接关系到kafka的吞吐量问题.kafka并没有提供太多高超的技巧;对于producer端,可以将消息buffer起来,当消息的条数达到一定阀值时,批量发送给broker;对于consumer端也是一样,批量fetch多条消息.不过消息量的大小可以通过配置文件来指定.对于kafka broker端,似乎有个sendfile系统调用可以潜在的提升网络IO的性能:将文件的数据映射到系统内存中,socket直接读取相应的内存区域即可,而无需进程再次copy和交换(这里涉及到"磁盘IO数据"/"内核内存"/"进程内存"/"网络缓冲区",多者之间的数据copy).

其实对于producer/consumer/broker三者而言,CPU的开支应该都不大,因此启用消息压缩机制是一个良好的策略;压缩需要消耗少量的CPU资源,不过对于kafka而言,网络IO更应该需要考虑.可以将任何在网络上传输的消息都经过压缩.kafka支持gzip/snappy等多种压缩方式.

**- 负载均衡**

kafka集群中的任何一个broker,都可以向producer提供metadata信息,这些metadata中包含"集群中存活的servers列表"/"partitions leader列表"等信息(请参看zookeeper中的节点信息). 当producer获取到metadata信息之后, producer将会和Topic下所有partition leader保持socket连接;消息由producer直接通过socket发送到broker,中间不会经过任何"路由层".

异步发送，将多条消息暂且在客户端buffer起来,并将他们批量发送到broker;小数据IO太多,会拖慢整体的网络延迟,批量延迟发送事实上提升了网络效率;不过这也有一定的隐患,比如当producer失效时,那些尚未发送的消息将会丢失。

**- 消息传输一致**

Kafka提供3种消息传输一致性语义：最多1次，最少1次，恰好1次。

最少1次：可能会重传数据，有可能出现数据被重复处理的情况;

最多1次：可能会出现数据丢失情况;

恰好1次：并不是指真正只传输1次，只不过有一个机制。确保不会出现“数据被重复处理”和“数据丢失”的情况。

**- 副本**

kafka中,replication策略是基于partition,而不是topic;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就像一个"consumer",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,这种同步策略,就要求follower和leader之间必须具有良好的网络环境.

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.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",partition leader较少的broker将会更有可能成为新的leader.

**- log**

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.

获取消息时,需要指定offset和最大chunk尺寸,offset用来表示消息的起始位置,chunk size用来表示最大获取消息的总长度(间接的表示消息的条数).根据offset,可以找到此消息所在segment文件,然后根据segment的最小offset取差值,得到它在file中的相对位置,直接读取输出即可.

**-分布式**

1) Producer端使用zookeeper用来"发现"broker列表,以及和Topic下每个partition leader建立socket连接并发送消息.

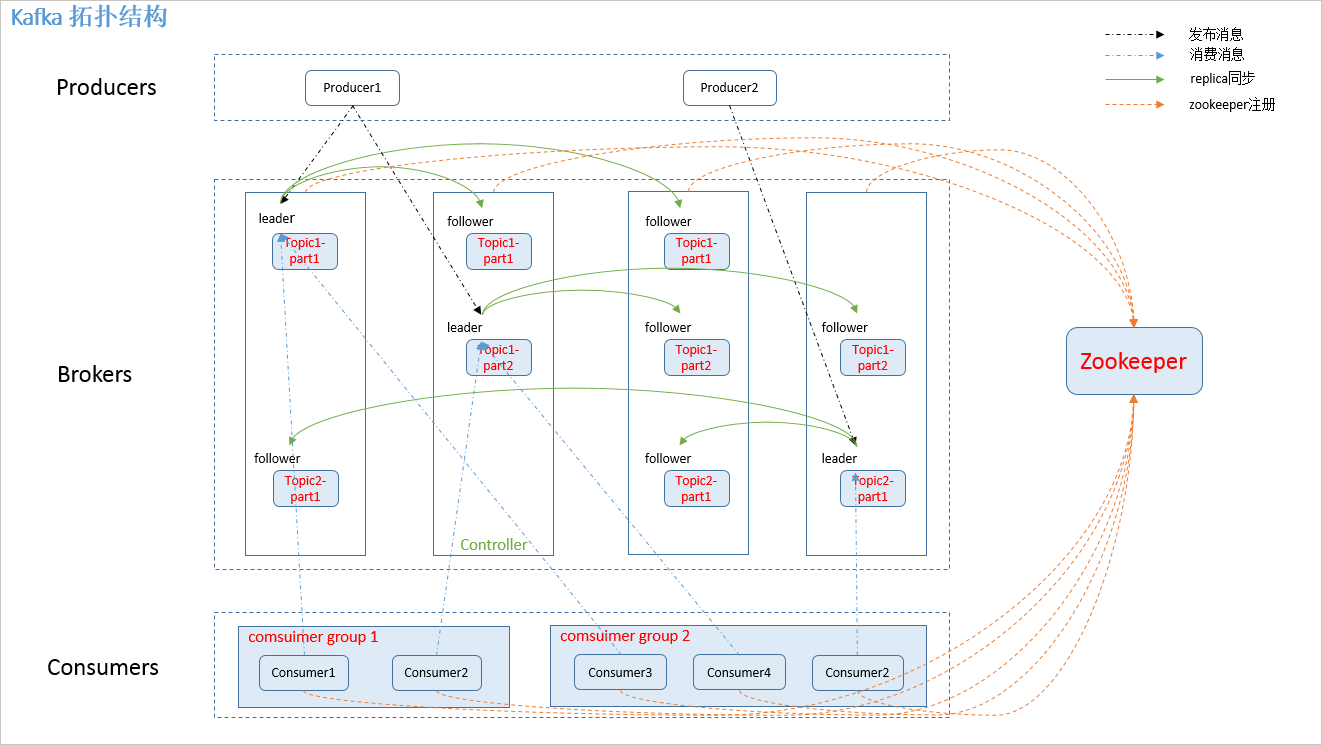
2) Broker端使用zookeeper用来注册broker信息,已经监测partition leader存活性.

3) Consumer端使用zookeeper用来注册consumer信息,其中包括consumer消费的partition列表等,同时也用来发现broker列表,并和partition leader建立socket连接,并获取消息。

## 二、深入知识

**1 kafka 架构**

1.1 拓扑结构



1.2 相关概念

1.producer：

　　消息生产者，发布消息到 kafka 集群的终端或服务。

2.broker：

　　kafka 集群中包含的服务器。

3.topic：

　　每条发布到 kafka 集群的消息属于的类别，即 kafka 是面向 topic 的。

4.partition：

　　partition 是物理上的概念，每个 topic 包含一个或多个 partition。kafka 分配的单位是 partition。

5.consumer：

　　从 kafka 集群中消费消息的终端或服务。

6.Consumer group：

　　high-level consumer API 中，每个 consumer 都属于一个 consumer group，每条消息只能被 consumer group 中的一个 Consumer 消费，但可以被多个 consumer group 消费。

7.replica：

　　partition 的副本，保障 partition 的高可用。

8.leader：

　　replica 中的一个角色， producer 和 consumer 只跟 leader 交互。

9.follower：

　　replica 中的一个角色，从 leader 中复制数据。

10.controller：

　　kafka 集群中的其中一个服务器，用来进行 leader election 以及 各种 failover。

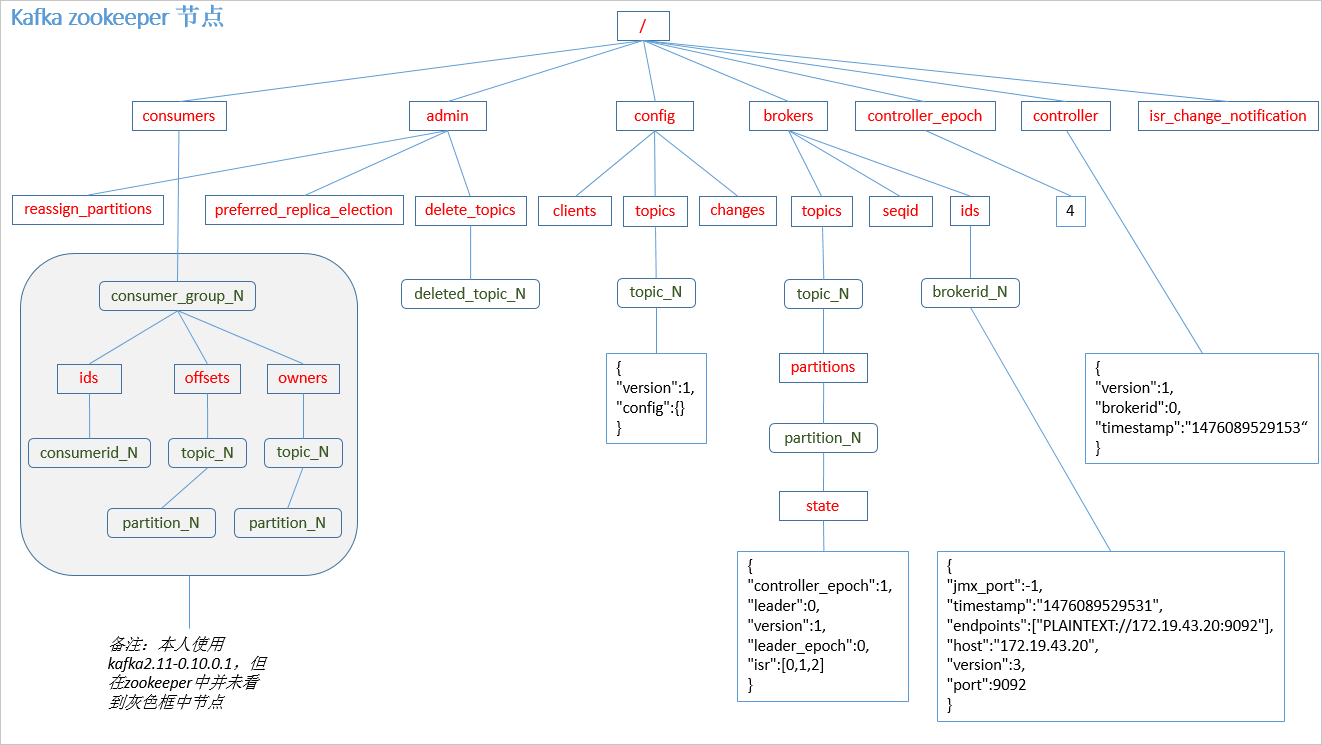
11.zookeeper：

kafka 通过 zookeeper 来存储集群的 meta 信息。

* **Broker**  
  Kafka集群包含一个或多个服务器，这种服务器被称为broker
* **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）
* **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
* **Producer**  
  负责发布消息到Kafka broker
* **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。

1.3 zookeeper 节点

kafka 在 zookeeper 中的存储结构如下图所示：



**2 producer 发布消息**

**2.1 写入方式**

producer 采用 push 模式将消息发布到 broker，每条消息都被 append 到 patition 中，属于顺序写磁盘（顺序写磁盘效率比随机写内存要高，保障 kafka 吞吐率）。

**2.2 消息路由**

producer 发送消息到 broker 时，会根据分区算法选择将其存储到哪一个 partition。其路由机制为：

1. 指定了 patition，则直接使用；

2. 未指定 patition 但指定 key，通过对 key 的 value 进行hash 选出一个 patition

2. patition 和 key 都未指定，使用轮询选出一个 patition。

附上 java 客户端分区源码，一目了然：

**//创建消息实例**

public ProducerRecord(String topic, Integer partition, Long timestamp, K key, V value) {

if (topic == null)

throw new IllegalArgumentException("Topic cannot be null");

if (timestamp != null && timestamp < 0)

throw new IllegalArgumentException("Invalid timestamp " + timestamp);

this.topic = topic;

this.partition = partition;

this.key = key;

this.value = value;

this.timestamp = timestamp;

}

**//计算 patition，如果指定了 patition 则直接使用，否则使用 key 计算**

private int partition(ProducerRecord<K, V> record, byte[] serializedKey , byte[] serializedValue, Cluster cluster) {

Integer partition = record.partition();

if (partition != null) {

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

int lastPartition = partitions.size() - 1;

if (partition < 0 || partition > lastPartition) {

throw new IllegalArgumentException(String.format("Invalid partition given with record: %d is not in the range [0...%d].", partition, lastPartition));

}

return partition;

}

return this.partitioner.partition(record.topic(), record.key(), serializedKey, record.value(), serializedValue, cluster);

}

**// 使用 key 选取 patition**

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

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

int numPartitions = partitions.size();

if (keyBytes == null) {

int nextValue = counter.getAndIncrement();

List<PartitionInfo> availablePartitions = cluster.availablePartitionsForTopic(topic);

if (availablePartitions.size() > 0) {

int part = DefaultPartitioner.toPositive(nextValue) % availablePartitions.size();

return availablePartitions.get(part).partition();

} else {

return DefaultPartitioner.toPositive(nextValue) % numPartitions;

}

} else {

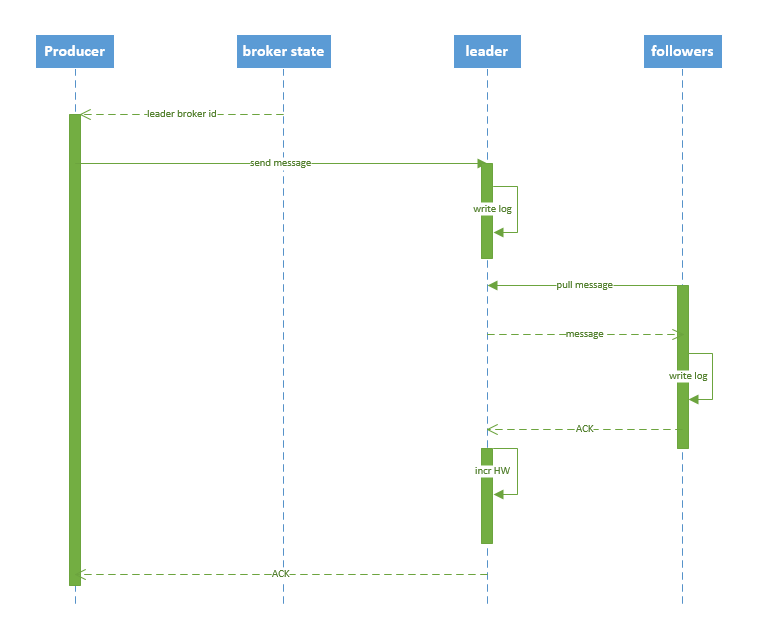
//对 keyBytes 进行 hash 选出一个 patition

return DefaultPartitioner.toPositive(Utils.murmur2(keyBytes)) % numPartitions;

}

}

**2.3 写入流程 producer 写入消息序列图如下所示：**



1. producer 先从 zookeeper 的 "/brokers/.../state" 节点找到该 partition 的 leader

2. producer 将消息发送给该 leader

2. leader 将消息写入本地 log

4. followers 从 leader pull 消息，写入本地 log 后 leader 发送 ACK

5. leader 收到所有 ISR 中的 replica 的 ACK 后，增加 HW（high watermark，最后 commit 的 offset） 并向 producer 发送 ACK

**2.4 producer delivery guarantee**

一般情况下存在三种情况：

*1. At most once 消息可能会丢，但绝不会重复传输*

*2. At least one 消息绝不会丢，但可能会重复传输*

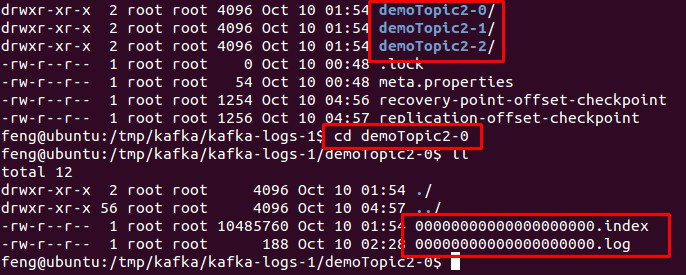
*2. Exactly once 每条消息肯定会被传输一次且仅传输一次*

当 producer 向 broker 发送消息时，一旦这条消息被 commit，由于 replication 的存在，它就不会丢。但是如果 producer 发送数据给 broker 后，遇到网络问题而造成通信中断，那 Producer 就无法判断该条消息是否已经 commit。虽然 Kafka 无法确定网络故障期间发生了什么，但是 producer 可以生成一种类似于主键的东西，发生故障时幂等性的重试多次，这样就做到了 Exactly once，但目前还并未实现。所以目前默认情况下一条消息从 producer 到 broker 是确保了 At least once，可通过设置 producer 异步发送实现At most once。

**3 broker 保存消息**

3.1 存储方式

物理上把 topic 分成一个或多个 patition（对应 server.properties 中的 num.partitions=3 配置），每个 patition 物理上对应一个文件夹（该文件夹存储该 patition 的所有消息和索引文件），如下：



3.2 存储策略

无论消息是否被消费，kafka 都会保留所有消息。有两种策略可以删除旧数据：

1. 基于时间：log.retention.hours=168

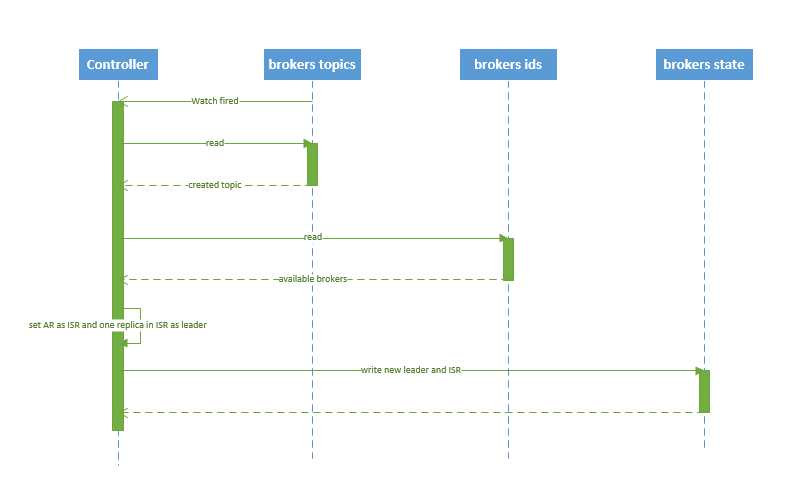
2. 基于大小：log.retention.bytes=1073741824

需要注意的是，因为Kafka读取特定消息的时间复杂度为O(1)，即与文件大小无关，所以这里删除过期文件与提高 Kafka 性能无关。

3.3 topic 创建与删除

3.3.1 创建 topic

创建 topic 的序列图如下所示：



1. controller 在 ZooKeeper 的 /brokers/topics 节点上注册 watcher，当 topic 被创建，则 controller 会通过 watch 得到该 topic 的 partition/replica 分配。

2. controller从 /brokers/ids 读取当前所有可用的 broker 列表，对于 set\_p 中的每一个 partition：

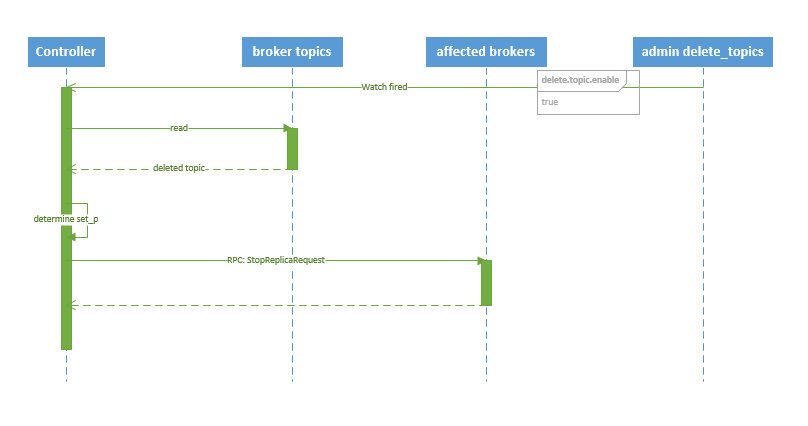
2.1 从分配给该 partition 的所有 replica（称为AR）中任选一个可用的 broker 作为新的 leader，并将AR设置为新的 ISR

2.2 将新的 leader 和 ISR 写入 /brokers/topics/[topic]/partitions/[partition]/state

3. controller 通过 RPC 向相关的 broker 发送 LeaderAndISRRequest。

3.3.2 删除 topic

删除 topic 的序列图如下所示：



1. controller 在 zooKeeper 的 /brokers/topics 节点上注册 watcher，当 topic 被删除，则 controller 会通过 watch 得到该 topic 的 partition/replica 分配。

2. 若 delete.topic.enable=false，结束；否则 controller 注册在 /admin/delete\_topics 上的 watch 被 fire，controller 通过回调向对应的 broker 发送 StopReplicaRequest。

**创建Topic时直接指定Topic Partition Replica与Kafka Broker之间的存储映射关系**

/usr/lib/kafka\_2.10-0.8.2.1/bin/kafka-topics.sh --zookeeper ZooKeeperHost:ZooKeeperPort --create --topic TopicName --replica-assignment id0:id1:id2,id3:id4:id5,id6:id7:id8

其中，“id0:id1:id2,id3:id4:id5,id6:id7:id8”**表示Topic TopicName一共有3个Partition（以“,”分隔），每个Partition均有3个Replica（以“:”分隔）**，Topic Partition Replica与Kafka Broker之间的对应关系如下：

Partition0 Replica：Broker id0、Broker id1、Broker id2；

Partition1 Replica：Broker id3、Broker id4、Broker id5；

Partition2 Replica：Broker id6、Broker id7、Broker id8；

**需要在这些 replica 之间选出一个 leader，producer 和 consumer 只与这个 leader 交互，其它 replica 作为 follower 从 leader 中复制数据。**

**4 kafka HA**

4.1 replication

如图.1所示，同一个 partition 可能会有多个 replica（对应 server.properties 配置中的 default.replication.factor=N）。没有 replica 的情况下，一旦 broker 宕机，其上所有 patition 的数据都不可被消费，同时 producer 也不能再将数据存于其上的 patition。引入replication 之后，同一个 partition 可能会有多个 replica，而这时需要在这些 replica 之间选出一个 leader，producer 和 consumer 只与这个 leader 交互，其它 replica 作为 follower 从 leader 中复制数据。

Kafka 分配 Replica 的算法如下：

1. 将所有 broker（假设共 n 个 broker）和待分配的 partition 排序

2. 将第 i 个 partition 分配到第（i mod n）个 broker 上

3. 将第 i 个 partition 的第 j 个 replica 分配到第（(i + j) mode n）个 broker上

4.2 leader failover

当 partition 对应的 leader 宕机时，需要从 follower 中选举出新 leader。在选举新leader时，一个基本的原则是，新的 leader 必须拥有旧 leader commit 过的所有消息。

kafka 在 zookeeper 中（/brokers/.../state）动态维护了一个 ISR（in-sync replicas），由3.3节的写入流程可知 ISR 里面的所有 replica 都跟上了 leader，只有 ISR 里面的成员才能选为 leader。对于 f+1 个 replica，一个 partition 可以在容忍 f 个 replica 失效的情况下保证消息不丢失。

当所有 replica 都不工作时，有两种可行的方案：

1. 等待 ISR 中的任一个 replica 活过来，并选它作为 leader。可保障数据不丢失，但时间可能相对较长。

2. 选择第一个活过来的 replica（不一定是 ISR 成员）作为 leader。无法保障数据不丢失，但相对不可用时间较短。

kafka 0.8.\* 使用第二种方式。

kafka 通过 Controller 来选举 leader，流程请参考4.3节。

4.3 broker failover

kafka broker failover 序列图如下所示：

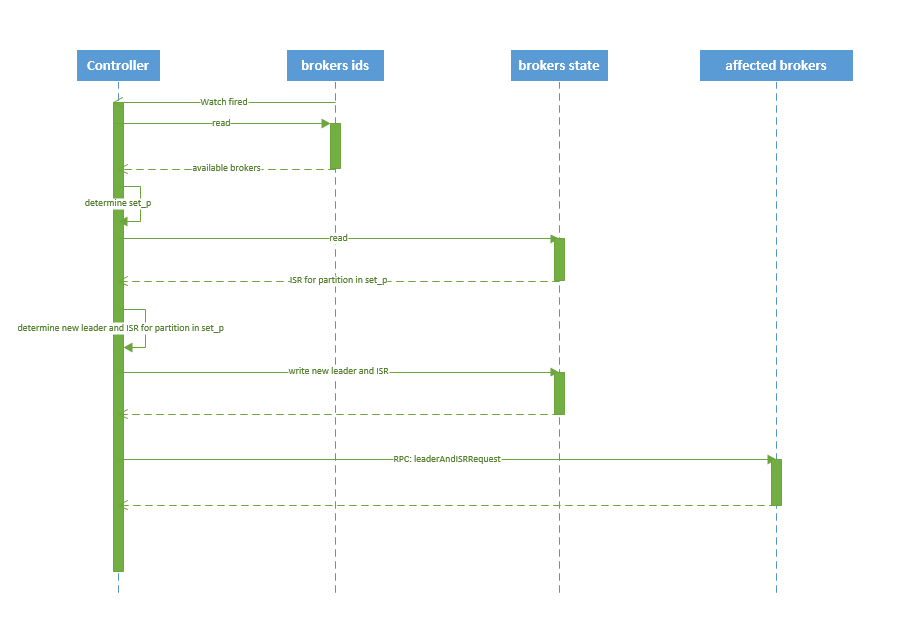


图.7

1. controller 在 zookeeper 的 /brokers/ids/[brokerId] 节点注册 Watcher，当 broker 宕机时 zookeeper 会 fire watch

2. controller 从 /brokers/ids 节点读取可用broker

3. controller决定set\_p，该集合包含宕机 broker 上的所有 partition

4. 对 set\_p 中的每一个 partition

4.1 从/brokers/topics/[topic]/partitions/[partition]/state 节点读取 ISR

4.2 决定新 leader（如4.3节所描述）

4.3 将新 leader、ISR、controller\_epoch 和 leader\_epoch 等信息写入 state 节点

4. 通过 RPC 向相关 broker 发送 leaderAndISRRequest 命令

4.4 controller failover

当 controller 宕机时会触发 controller failover。每个 broker 都会在 zookeeper 的 "/controller" 节点注册 watcher，当 controller 宕机时 zookeeper 中的临时节点消失，所有存活的 broker 收到 fire 的通知，每个 broker 都尝试创建新的 controller path，只有一个竞选成功并当选为 controller。

当新的 controller 当选时，会触发 KafkaController.onControllerFailover 方法，在该方法中完成如下操作：

1. 读取并增加 Controller Epoch。

2. 在 reassignedPartitions Patch(/admin/reassign\_partitions) 上注册 watcher。

3. 在 preferredReplicaElection Path(/admin/preferred\_replica\_election) 上注册 watcher。

4. 通过 partitionStateMachine 在 broker Topics Patch(/brokers/topics) 上注册 watcher。

4. 若 delete.topic.enable=true（默认值是 false），则 partitionStateMachine 在 Delete Topic Patch(/admin/delete\_topics) 上注册 watcher。

6. 通过 replicaStateMachine在 Broker Ids Patch(/brokers/ids)上注册Watch。

7. 初始化 ControllerContext 对象，设置当前所有 topic，“活”着的 broker 列表，所有 partition 的 leader 及 ISR等。

8. 启动 replicaStateMachine 和 partitionStateMachine。

9. 将 brokerState 状态设置为 RunningAsController。

10. 将每个 partition 的 Leadership 信息发送给所有“活”着的 broker。

11. 若 auto.leader.rebalance.enable=true（默认值是true），则启动 partition-rebalance 线程。

12. 若 delete.topic.enable=true 且Delete Topic Patch(/admin/delete\_topics)中有值，则删除相应的Topic。

**5 consumer 消费消息**

**5.1 consumer API**

kafka 提供了两套 consumer API：

1. The high-level Consumer API

2. The SimpleConsumer API

其中 high-level consumer API 提供了个从 kafka 消费数据的高层抽象，而 SimpleConsumer API 则需要开发人员更多地关注细节。

5.1.1 The high-level consumer API

high-level consumer API 提供了 consumer group 的语义，一个消息只能被 group 内的一个 consumer 所消费，且 consumer 消费消息时不关注 offset，最后一个 offset 由 zookeeper 保存。

使用 high-leve consumer API 可以是多线程的应用，应当注意：

1. 如果消费线程大于 patition 数量，则有些线程将收不到消息

2. 如果 patition 数量大于线程数，则有些线程多收到多个 patition 的消息

3. 如果一个线程消费多个 patition，则无法保证你收到的消息的顺序，而一个 patition 内的消息是有序的

5.1.2 The SimpleConsumer API

如果你想要对 patition 有更多的控制权，那就应该使用 SimpleConsumer API，比如：

1. 多次读取一个消息

2. 只消费一个 patition 中的部分消息

3. 使用事务来保证一个消息仅被消费一次

但是使用此 API 时，partition、offset、broker、leader 等对你不再透明，需要自己去管理。你需要做大量的额外工作：

1. 必须在应用程序中跟踪 offset，从而确定下一条应该消费哪条消息

2. 应用程序需要通过程序获知每个 Partition 的 leader 是谁

3. 需要处理 leader 的变更

使用 SimpleConsumer API 的一般流程如下：

1. 查找到一个“活着”的 broker，并且找出每个 partition 的 leader

2. 找出每个 partition 的 follower

3. 定义好请求，该请求应该能描述应用程序需要哪些数据

4. fetch 数据

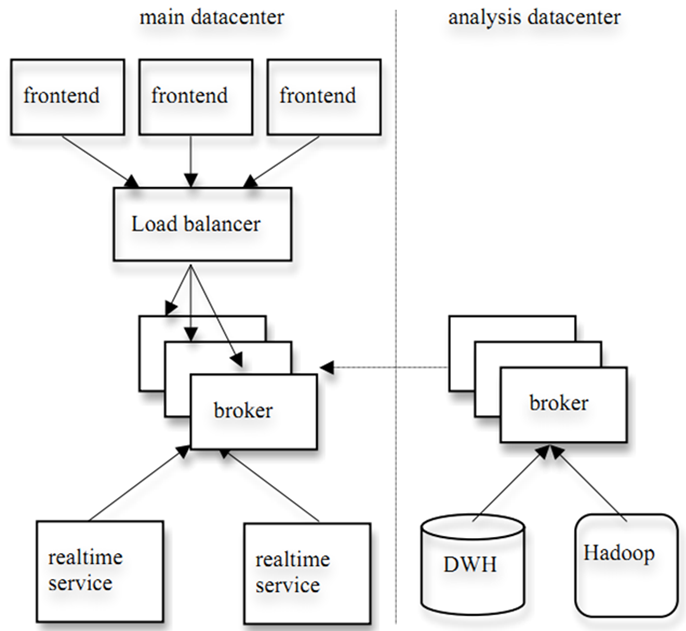
5. 识别 leader 的变化，并对之作出必要的响应

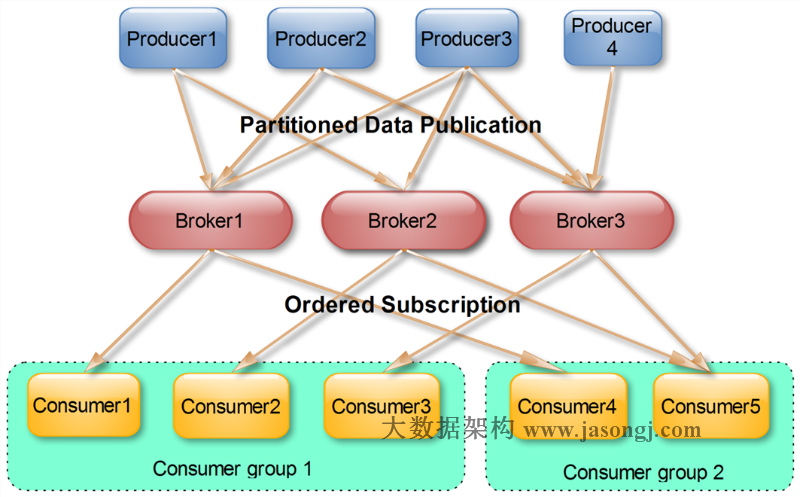
以下针对 high-level Consumer API 进行说明。

**5.2 consumer group**

如 2.2 节所说， kafka 的分配单位是 patition。每个 consumer 都属于一个 group，一个 partition 只能被同一个 group 内的一个 consumer 所消费（也就保障了一个消息只能被 group 内的一个 consuemr 所消费），但是多个 group 可以同时消费这个 partition。

kafka 的设计目标之一就是同时实现离线处理和实时处理，根据这一特性，可以使用 spark/Storm 这些实时处理系统对消息在线处理，同时使用 Hadoop 批处理系统进行离线处理，还可以将数据备份到另一个数据中心，只需要保证这三者属于不同的 consumer group。如下图所示：





**5.3 消费方式**

consumer 采用 pull 模式从 broker 中读取数据。

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。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。

对于 Kafka 而言，pull 模式更合适，它可简化 broker 的设计，consumer 可自主控制消费消息的速率，同时 consumer 可以自己控制消费方式——即可批量消费也可逐条消费，同时还能选择不同的提交方式从而实现不同的传输语义。

**5.4 consumer delivery guarantee**

如果将 consumer 设置为 autocommit，consumer 一旦读到数据立即自动 commit。如果只讨论这一读取消息的过程，那 Kafka 确保了 Exactly once。

但实际使用中应用程序并非在 consumer 读取完数据就结束了，而是要进行进一步处理，而数据处理与 commit 的顺序在很大程度上决定了consumer delivery guarantee：

1.读完消息先 commit 再处理消息。

这种模式下，如果 consumer 在 commit 后还没来得及处理消息就 crash 了，下次重新开始工作后就无法读到刚刚已提交而未处理的消息，这就对应于 At most once

2.读完消息先处理再 commit。

这种模式下，如果在处理完消息之后 commit 之前 consumer crash 了，下次重新开始工作时还会处理刚刚未 commit 的消息，实际上该消息已经被处理过了。这就对应于 At least once。

3.如果一定要做到 Exactly once，就需要协调 offset 和实际操作的输出。

精典的做法是引入两阶段提交。如果能让 offset 和操作输入存在同一个地方，会更简洁和通用。这种方式可能更好，因为许多输出系统可能不支持两阶段提交。比如，consumer 拿到数据后可能把数据放到 HDFS，如果把最新的 offset 和数据本身一起写到 HDFS，那就可以保证数据的输出和 offset 的更新要么都完成，要么都不完成，间接实现 Exactly once。（目前就 high-level API而言，offset 是存于Zookeeper 中的，无法存于HDFS，而SimpleConsuemr API的 offset 是由自己去维护的，可以将之存于 HDFS 中）

总之，Kafka 默认保证 At least once，并且允许通过设置 producer 异步提交来实现 At most once（见文章《kafka consumer防止数据丢失》）。而 Exactly once 要求与外部存储系统协作，幸运的是 kafka 提供的 offset 可以非常直接非常容易得使用这种方式。

**5.5 consumer rebalance**

当有 consumer 加入或退出、以及 partition 的改变（如 broker 加入或退出）时会触发 rebalance。consumer rebalance算法如下：

1. 将目标 topic 下的所有 partirtion 排序，存于PT

2. 对某 consumer group 下所有 consumer 排序，存于 CG，第 i 个consumer 记为 Ci

3. N=size(PT)/size(CG)，向上取整

4. 解除 Ci 对原来分配的 partition 的消费权（i从0开始）

5. 将第i\*N到（i+1）\*N-1个 partition 分配给 Ci

在 0.8.\*版本，每个 consumer 都只负责调整自己所消费的 partition，为了保证整个consumer group 的一致性，当一个 consumer 触发了 rebalance 时，该 consumer group 内的其它所有其它 consumer 也应该同时触发 rebalance。这会导致以下几个问题：

1.Herd effect

任何 broker 或者 consumer 的增减都会触发所有的 consumer 的 rebalance

2.Split Brain

每个 consumer 分别单独通过 zookeeper 判断哪些 broker 和 consumer 宕机了，那么不同 consumer 在同一时刻从 zookeeper 看到的 view 就可能不一样，这是由 zookeeper 的特性决定的，这就会造成不正确的 reblance 尝试。

3. 调整结果不可控

所有的 consumer 都并不知道其它 consumer 的 rebalance 是否成功，这可能会导致 kafka 工作在一个不正确的状态。

在Kafka在0.8以前的版本中，是没有Replication的，一旦某一个Broker宕机，则其上所有的Partition数据都不可被消费，这与Kafka数据持久性及Delivery Guarantee的设计目标相悖。同时Producer都不能再将数据存于这些Partition中。

如果Producer使用同步模式则Producer会在尝试重新发送message.send.max.retries（默认值为3）次后抛出Exception，用户可以选择停止发送后续数据也可选择继续选择发送。而前者会造成数据的阻塞，后者会造成本应发往该Broker的数据的丢失。

如果Producer使用异步模式，则Producer会尝试重新发送message.send.max.retries（默认值为3）次后记录该异常并继续发送后续数据，这会造成数据丢失并且用户只能通过日志发现该问题。

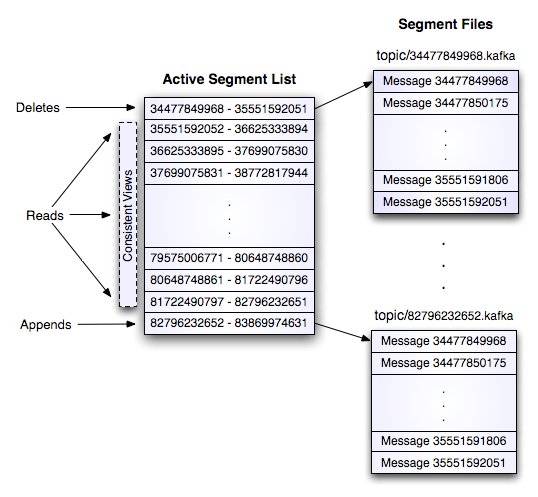
由此可见，在没有Replication的情况下，一旦某机器宕机或者某个Broker停止工作则会造成整个系统的可用性降低。随着集群规模的增加，整个集群中出现该类异常的几率大大增加，因此对于生产系统而言Replication机制的引入非常重要。

**6 kafka存储**

Kafka的存储布局非常简单。话题的每个分区对应一个逻辑日志。物理上，一个日志为相同大小的一组分段文件。每次生产者发布消息到一个分区，代理就将消息追加到最后一个段文件中。当发布的消息数量达到设定值或者经过一定的时间后，段文件真正写入磁盘中。写入完成后，消息公开给消费者。

与传统的消息系统不同，Kafka系统中存储的消息没有明确的消息Id。消息通过日志中的逻辑偏移量来公开。这样就避免了维护配套密集寻址，用于映射消息ID到实际消息地址的随机存取索引结构的开销。消息偏移量是增量的，但不连续。要计算下一消息的偏移量，可以在其逻辑偏移的基础上加上当前消息的长度。

消费者始终从特定分区顺序地获取消息，如果消费者知道特定消息的偏移量，也就说明消费者已经消费了之前的所有消息。消费者向代理发出异步拉请求，准备字节缓冲区用于消费。每个异步拉请求都包含要消费的消息偏移量。Kafka利用sendfile API高效地从代理的日志段文件中分发字节给消费者。



**7代理**

不同于其他消息系统，kafka代理是无状态的，即消费者必须维护已消费的状态消息，而代理完全不管。

这种设计的创新在于:

·代理以一个基于时间的SLA应用于保留策略。当消息在代理中超过一定时间后，将会被自动删除。

·消费者可以故意倒回到老的偏移量再次消费数据。虽然这违法了队列的常见约定，但常见于许多业务中。

**8与zookeeper的关系**

kafka使用ZooKeeper用于管理、协调代理。每个Kafka代理通过Zookeeper协调其他Kafka代理。

当Kafka系统中新增了代理或某个代理失效时，Zookeeper服务将通知生产者和消费者。生产者与消费者据此开始与其他代理协调工作。

Zookeeper在Kakfa中扮演的角色：Kafka将元数据信息保存在Zookeeper中，但是发送给Topic本身的数据是不会发到Zk上的

@ kafka使用zookeeper来实现动态的集群扩展，不需要更改客户端（producer和consumer）的配置。broker会在zookeeper注册并保持相关的元数据（topic，partition信息等）更新。

@ 而客户端会在zookeeper上注册相关的watcher。一旦zookeeper发生变化，客户端能及时感知并作出相应调整。这样就保证了添加或去除broker时，各broker间仍能自动实现负载均衡。这里的客户端指的是Kafka的消息生产端(Producer)和消息消费端(Consumer)

@ Broker端使用zookeeper来注册broker信息,以及监测partitionleader存活性.

@ Consumer端使用zookeeper用来注册consumer信息,其中包括consumer消费的partition列表等,同时也用来发现broker列表,并和partitionleader建立socket连接,并获取消息.

@ Zookeer和Producer没有建立关系，只和Brokers、Consumers建立关系以实现负载均衡，即同一个ConsumerGroup中的Consumers可以实现负载均衡（因为Producer是瞬态的，可以发送后关闭，无需直接等待）

**8 kafka的设计**

**（1）、吞吐量**

高吞吐是kafka需要实现的核心目标之一，为此kafka做了以下一些设计：

1). 数据磁盘持久化：消息不在内存中cache，直接写入到磁盘，充分利用磁盘的顺序读写性能

2). zero-copy：减少IO操作步骤

3). 数据批量发送

4). 数据压缩

5). Topic划分为多个partition，提高parallelism（并行）

**（2）、负载均衡**

1). producer根据用户指定的算法，将消息发送到指定的partition

2). 存在多个partiiton，每个partition有自己的replica，每个replica分布在不同的Broker节点上

3). 多个partition需要选取出leadpartition，lead partition负责读写，并由zookeeper负责fail over

4). 通过zookeeper管理broker与consumer的动态加入与离开

**（3）、拉取系统**

由于kafka broker会持久化数据，broker没有内存压力，因此，consumer非常适合采取pull的方式消费数据，具有以下几点好处：

1). 简化kafka设计

2). consumer根据消费能力自主控制消息拉取速度

3). consumer根据自身情况自主选择消费模式，例如批量，重复消费，从尾端开始消费等

**（4）、可扩展性**

当需要增加broker结点时，新增的broker会向zookeeper注册，而producer及consumer会根据注册在zookeeper上的watcher感知这些变化，并及时作出调整。

（5）设计要点

1)、直接使用Linux 文件系统的cache，来高效缓存数据。

2)、采用linux Zero-Copy提高发送性能。传统的数据发送需要发送4次上下文切换，采用sendfile系统调用之后，数据直接在内核态交换，系统上下文切换减少为2次。根据测试结果，可以提高60%的数据发送性能。Zero-Copy详细的技术细节可以参考：https://www.ibm.com/developerworks/linux/library/j-zerocopy/

3)、数据在磁盘上存取代价为O(1)。kafka以topic来进行消息管理，每个topic包含多个part（ition），每个part对应一个逻辑log，有多个segment组成。每个segment中存储多条消息，消息id由其逻辑位置决定，即从消息id可直接定位到消息的存储位置，避免id到位置的额外映射。每个part在内存中对应一个index，记录每个segment中的第一条消息偏移。发布者发到某个topic的消息会被均匀的分布到多个part上（随机或根据用户指定的回调函数进行分布），broker收到发布消息往对应part的最后一个segment上添加该消息，当某个segment上的消息条数达到配置值或消息发布时间超过阈值时，segment上的消息会被flush到磁盘，只有flush到磁盘上的消息订阅者才能订阅到，segment达到一定的大小后将不会再往该segment写数据，broker会创建新的segment。

4)、显式分布式，即所有的producer、broker和consumer都会有多个，均为分布式的。Producer和broker之间没有负载均衡机制。broker和consumer之间利用zookeeper进行负载均衡。所有broker和consumer都会在zookeeper中进行注册，且zookeeper会保存他们的一些元数据信息。如果某个broker和consumer发生了变化，所有其他的broker和consumer都会得到通知。

**9 [kafka java示例](http://blog.csdn.net/csolo/article/details/52447392)**

kafka作为分布式日志收集或系统监控服务，我们有必要在合适的场合使用它。kafka的部署包括zookeeper环境/kafka环境，同时还需要进行一些配置操作.接下来介绍如何使用kafka.

我们使用3个zookeeper实例构建zk集群，使用2个kafka broker构建kafka集群.

其中kafka为0.8V，zookeeper为3.4.5V

**（1）.Zookeeper集群构建**

我们有3个zk实例，分别为zk-0,zk-1,zk-2;如果你仅仅是测试使用，可以使用1个zk实例.

1) zk-0

调整配置文件：

Php代码 收藏代码

clientPort=2181

server.0=127.0.0.1:2888:3888

server.1=127.0.0.1:2889:3889

server.2=127.0.0.1:2890:3890

##只需要修改上述配置，其他配置保留默认值

启动zookeeper

./zkServer.sh start

2) zk-1

调整配置文件(其他配置和zk-0一只)：

clientPort=2182

##只需要修改上述配置，其他配置保留默认值

启动zookeeper

./zkServer.sh start

3) zk-2

调整配置文件(其他配置和zk-0一只)：

clientPort=2183

##只需要修改上述配置，其他配置保留默认值

启动zookeeper

./zkServer.sh start

**（2）Kafka集群构建**

因为Broker配置文件涉及到zookeeper的相关约定，因此我们先展示broker配置文件.我们使用2个kafka broker来构建这个集群环境，分别为kafka-0,kafka-1.

1) kafka-0

在config目录下修改配置文件为：

broker.id=0

port=9092

num.network.threads=2

num.io.threads=2

socket.send.buffer.bytes=1048576

socket.receive.buffer.bytes=1048576

socket.request.max.bytes=104857600

log.dir=./logs

num.partitions=2

log.flush.interval.messages=10000

log.flush.interval.ms=1000

log.retention.hours=168

#log.retention.bytes=1073741824

log.segment.bytes=536870912

##replication机制,让每个topic的partitions在kafka-cluster中备份2个

##用来提高cluster的容错能力..

default.replication.factor=1

log.cleanup.interval.mins=10

zookeeper.connect=127.0.0.1:2181,127.0.0.1:2182,127.0.0.1:2183

zookeeper.connection.timeout.ms=1000000

因为kafka用Scala语言编写，因此运行kafka需要首先准备scala相关环境。

> cd kafka-0

> ./sbt update

> ./sbt package

> ./sbt assembly-package-dependency

其中最后一条指令执行有可能出现异常，暂且不管。 启动kafka broker：

> JMS\_PORT=9997 bin/kafka-server-start.sh config/server.properties &

因为zookeeper环境已经正常运行了，我们无需通过kafka来挂载启动zookeeper.如果你的一台机器上部署了多个kafka broker，你需要声明JMS\_PORT.

2) kafka-1

broker.id=1

port=9093

##其他配置和kafka-0保持一致

然后和kafka-0一样执行打包命令，然后启动此broker.

> JMS\_PORT=9998 bin/kafka-server-start.sh config/server.properties &

仍然可以通过如下指令查看topic的"partition"/"replicas"的分布和存活情况.

> bin/kafka-list-topic.sh --zookeeper localhost:2181

topic: my-replicated-topic partition: 0 leader: 2 replicas: 1,2,0 isr: 2

topic: test partition: 0 leader: 0 replicas: 0 isr: 0

**（3）Maven依赖**

<dependencies>

<dependency>

<groupId>log4j</groupId>

<artifactId>log4j</artifactId>

<version>1.2.14</version>

</dependency>

<dependency>

<groupId>org.apache.kafka</groupId>

<artifactId>kafka\_2.8.2</artifactId>

<version>0.8.0</version>

<exclusions>

<exclusion>

<groupId>log4j</groupId>

<artifactId>log4j</artifactId>

</exclusion>

</exclusions>

</dependency>

<dependency>

<groupId>org.scala-lang</groupId>

<artifactId>scala-library</artifactId>

<version>2.8.2</version>

</dependency>

<dependency>

<groupId>com.yammer.metrics</groupId>

<artifactId>metrics-core</artifactId>

<version>2.2.0</version>

</dependency>

<dependency>

<groupId>com.101tec</groupId>

<artifactId>zkclient</artifactId>

<version>0.3</version>

</dependency>

</dependencies>

**（4）Producer端代码**

1) producer.properties文件：此文件放在/resources目录下

#partitioner.class=

##broker列表可以为kafka server的子集,因为producer需要从broker中获取metadata

##尽管每个broker都可以提供metadata,此处还是建议,将所有broker都列举出来

metadata.broker.list=127.0.0.1:9092,127.0.0.1:9093

##,127.0.0.1:9093

##同步,建议为async

producer.type=sync

compression.codec=0

serializer.class=kafka.serializer.StringEncoder

##在producer.type=async时有效

#batch.num.messages=100

2) LogProducer.java代码样例

package com.test.kafka;

import java.util.ArrayList;

import java.util.Collection;

import java.util.List;

import java.util.Properties;

import kafka.javaapi.producer.Producer;

import kafka.producer.KeyedMessage;

import kafka.producer.ProducerConfig;

public class LogProducer {

private Producer<String,String> inner;

public LogProducer() throws Exception{

Properties properties = new Properties();

properties.load(ClassLoader.getSystemResourceAsStream("producer.properties"));

ProducerConfig config = new ProducerConfig(properties);

inner = new Producer<String, String>(config);

}

public void send(String topicName,String message) {

if(topicName == null || message == null){

return;

}

KeyedMessage<String, String> km = new KeyedMessage<String, String>(topicName,message);//如果具有多个partitions,请使用new KeyedMessage(String topicName,K key,V value).

inner.send(km);

}

public void send(String topicName,Collection<String> messages) {

if(topicName == null || messages == null){

return;

}

if(messages.isEmpty()){

return;

}

List<KeyedMessage<String, String>> kms = new ArrayList<KeyedMessage<String, String>>();

for(String entry : messages){

KeyedMessage<String, String> km = new KeyedMessage<String, String>(topicName,entry);

kms.add(km);

}

inner.send(kms);

}

public void close(){

inner.close();

}

/\*\*

\* @param args

\*/

public static void main(String[] args) {

LogProducer producer = null;

try{

producer = new LogProducer();

int i=0;

while(true){

producer.send("test-topic", "this is a sample" + i);

i++;

Thread.sleep(2000);

}

}catch(Exception e){

e.printStackTrace();

}finally{

if(producer != null){

producer.close();

}

}

}

}

**（5）Consumer端**

1) consumer.properties:文件位于/resources目录下

zookeeper.connect=127.0.0.1:2181,127.0.0.1:2182,127.0.0.1:2183

##,127.0.0.1:2182,127.0.0.1:2183

# timeout in ms for connecting to zookeeper

zookeeper.connectiontimeout.ms=1000000

#consumer group id

group.id=test-group

#consumer timeout

#consumer.timeout.ms=5000

auto.commit.enable=true

auto.commit.interval.ms=60000

2) LogConsumer.java代码样例

package com.test.kafka;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

import java.util.Properties;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import kafka.consumer.Consumer;

import kafka.consumer.ConsumerConfig;

import kafka.consumer.ConsumerIterator;

import kafka.consumer.KafkaStream;

import kafka.javaapi.consumer.ConsumerConnector;

import kafka.message.MessageAndMetadata;

public class LogConsumer {

private ConsumerConfig config;

private String topic;

private int partitionsNum;

private MessageExecutor executor;

private ConsumerConnector connector;

private ExecutorService threadPool;

public LogConsumer(String topic,int partitionsNum,MessageExecutor executor) throws Exception{

Properties properties = new Properties();

properties.load(ClassLoader.getSystemResourceAsStream("consumer.properties"));

config = new ConsumerConfig(properties);

this.topic = topic;

this.partitionsNum = partitionsNum;

this.executor = executor;

}

public void start() throws Exception{

connector = Consumer.createJavaConsumerConnector(config);

Map<String,Integer> topics = new HashMap<String,Integer>();

topics.put(topic, partitionsNum);

Map<String, List<KafkaStream<byte[], byte[]>>> streams = connector.createMessageStreams(topics);

List<KafkaStream<byte[], byte[]>> partitions = streams.get(topic);

threadPool = Executors.newFixedThreadPool(partitionsNum);

for(KafkaStream<byte[], byte[]> partition : partitions){

threadPool.execute(new MessageRunner(partition));

}

}

public void close(){

try{

threadPool.shutdownNow();

}catch(Exception e){

//

}finally{

connector.shutdown();

}

}

class MessageRunner implements Runnable{

private KafkaStream<byte[], byte[]> partition;

MessageRunner(KafkaStream<byte[], byte[]> partition) {

this.partition = partition;

}

public void run(){

ConsumerIterator<byte[], byte[]> it = partition.iterator();

while(it.hasNext()){

//connector.commitOffsets();手动提交offset,当autocommit.enable=false时使用

MessageAndMetadata<byte[],byte[]> item = it.next();

System.out.println("partiton:" + item.partition());

System.out.println("offset:" + item.offset());

executor.execute(new String(item.message()));//UTF-8,注意异常

}

}

}

interface MessageExecutor {

public void execute(String message);

}

/\*\*

\* @param args

\*/

public static void main(String[] args) {

LogConsumer consumer = null;

try{

MessageExecutor executor = new MessageExecutor() {

public void execute(String message) {

System.out.println(message);

}

};

consumer = new LogConsumer("test-topic", 2, executor);

consumer.start();

}catch(Exception e){

e.printStackTrace();

}finally{

// if(consumer != null){

// consumer.close();

// }

}

}

}

需要提醒的是,上述LogConsumer类中,没有太多的关注异常情况,必须在MessageExecutor.execute()方法中抛出异常时的情况.

在测试时，建议优先启动consumer，然后再启动producer，这样可以实时的观测到最新的消息。

## 三、优点

1.主要是用来解决百万级别的数据中生产者和消费者之间数据传输的问题

2.可以将一条数据提供给多个接收这做不同的处理

3.当两个系统是隔绝的，无法通信的时候，如果想要他们通信就需要重新构建其中的一个工程，而kafka实现了生产者和消费者之间的无缝对接。

## 四、使用场景

1.消息队列

比起大多数的消息系统来说，Kafka有更好的吞吐量，内置的分区，冗余及容错性，这让Kafka成为了一个很好的大规模消息处理应用的解决方案。消息系统一般吞吐量相对较低，但是需要更小的端到端延时，并尝尝依赖于Kafka提供的强大的持久性保障。在这个领域，Kafka足以媲美传统消息系统，如ActiveMR或RabbitMQ。

2.行为跟踪

Kafka的另一个应用场景是跟踪用户浏览页面、搜索及其他行为，以发布-订阅的模式实时记录到对应的topic里。那么这些结果被订阅者拿到后，就可以做进一步的实时处理，或实时监控，或放到Hadoop/离线数据仓库里处理。

3.元信息监控

作为操作记录的监控模块来使用，即汇集记录一些操作信息，可以理解为运维性质的数据监控吧。

4.日志收集

使用Kafka代替日志聚合（logaggregation）。日志聚合一般来说是从服务器上收集日志文件，然后放到一个集中的位置（文件服务器或HDFS）进行处理。然而Kafka忽略掉文件的细节，将其更清晰地抽象成一个个日志或事件的消息流。这就让Kafka处理过程延迟更低，更容易支持多数据源和分布式数据处理。比起以日志为中心的系统比如Scribe或者Flume来说，Kafka提供同样高效的性能和因为复制导致的更高的耐用性保证，以及更低的端到端延迟。

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

5.流处理

这个场景可能比较多，也很好理解。保存收集流数据，以提供之后对接的Storm或其他流式计算框架进行处理。很多用户会将那些从原始topic来的数据进行阶段性处理，汇总，扩充或者以其他的方式转换到新的topic下再继续后面的处理。例如一个文章推荐的处理流程，可能是先从RSS数据源中抓取文章的内容，然后将其丢入一个叫做“文章”的topic中；后续操作可能是需要对这个内容进行清理，比如回复正常数据或者删除重复数据，最后再将内容匹配的结果返还给用户。这就在一个独立的topic之外，产生了一系列的实时数据处理的流程。Strom和Samza是非常著名的实现这种类型数据转换的框架。

6.事件源

事件源是一种应用程序设计的方式，该方式的状态转移被记录为按时间顺序排序的记录序列。Kafka可以存储大量的日志数据，这使得它成为一个对这种方式的应用来说绝佳的后台。比如动态汇总（News feed）。

7.持久性日志（commit log）

Kafka可以为一种外部的持久性日志的分布式系统提供服务。这种日志可以在节点间备份数据，并为故障节点数据回复提供一种重新同步的机制。Kafka中日志压缩功能为这种用法提供了条件。在这种用法中，Kafka类似于Apache BookKeeper项目。