**1 血统纯正的ActiveMQ**  
AMQ是企业级应用的宠儿，Apache出品，完全遵循JMS规范，但是又非常的“重”，在互联网中它的性能令人诟病，目前（5.9版本），它依托Zookeeper和LevelDB支持主备failover切换及副本备份，但负载均衡还是不够完善……下面我们分话题来谈：  
1）消息的安全性  
在集群环境下，通过Zookeeper选出master，其余节点均为slave，只有master接受和处理客户端连接，其余slave节点均连接至master，并同步所有的持久化操作。（但是此种模式不支持延迟消息和计划消息，因为他们存储在KahaDB文件里。）如果master挂了，最后更新的slave将被选中当master。  
并且，所有的消息操作需要集群一定节点(N/2+1)都操作成功才会返回操作成功的结果，如果是N=3个replicas节点，那么需要集群中有3/2+1=2个节点都确认操作成功。  
所以如果结合JMS规范的ACK指令，消息的安全性在一个健康的集群中完全可以得到保障，三个节点的集群可以忍受1个节点挂掉仍可以提供服务。  
2）服务的稳定容错性  
客户端通过failover协议连接集群服务，所以AMQ的稳定性主要依赖于ZK和自身集群的节点状态，ZK集群的稳定性暂且不表，AMQ自身集群主要是通过公式(N/2+1)来计算，上一节已经谈到3个节点允许挂一个节点，看起来稳定性还是可以的，唯一担心的就是客户端的读写链接压力全部在master上。  
3） 吞吐量  
目前主备集群中所有的topic和queue的读写都在单节点的master上进行，所以吞吐量完全依赖单点master，看官网上AMQ是支持Broker clusters的，但是这个支持缺点很多，首先它依然采用failover协议，这看起来与主备集群冲突了，第二点如果其中一个broker挂了，那它所属的还没有被消费的消息将无法消费直到broker重启成功。  
如果将Broker集群与主备集群结合起来倒是满足去除单点又达到负载均衡的需求，可惜两者都通过failover协议连接，目测需要改造后才能使用。  
  
**最后我们看两个AMQ的特性：**  
1）Virtual Topics  
同一个应用上的一个topic订阅不能使用多个consumer来共同承担消息处理功能。因为每个consume都会获取所有消息，而且如果这个consume挂了，消息就无法被消费了，使用Virtual Topics可以实现消费端分组，实际上存储的时候是被转换成Queue的语义存储的。这样多个消费者可以同时消费同一个虚拟topic又不会收到全部的消息  
2）Composite Destinations  
这个特性的意义主要就是方便吧，可以同时发送一份消息至多个topic和多个queue，还可以使用filter去select某些消息的方向。具体参考官网文档吧。

01 性能怪兽Kafka

Kafka是LinkedIn开源的分布式发布-订阅消息系统,目前归属于Apache定级项目。”Apache Kafka is publish-subscribe messaging rethought as a distributed commit log.”,官网首页的一句话高度概括其职责。Kafka并没有遵守JMS规范,他只用文件系统来管理消息的生命周期。Kafka的设计目标是:

(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复杂度的访问性能。

(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条以上消息的传输。

(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。

(5)Scale out:支持在线水平扩展。

所以,不像AMQ,Kafka从设计开始极为高可用为目的,天然HA。broker支持集群,消息亦支持负载均衡,还有副本机制。同样,Kafka也是使用Zookeeper管理集群节点信息,包括consumer的消费信息也是保存在zk中,下面我们分话题来谈:

1)消息的安全性

Kafka集群中的Leader负责某一topic的某一partition的消息的读写,理论上consumer和producer只与该Leader节点打交道,一个集群里的某一broker即是Leader的同时也可以担当某一partition的follower,即Replica。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

同时,Kafka与Replica既非同步也不是严格意义上的异步。一个典型的Kafka发送-消费消息的过程如下:首先首先Producer消息发送给某Topic的某Partition的Leader,Leader先是将消息写入本地Log,同时follower(如果落后过多将会被踢出出Replica列表)从Leader上pull消息,并且在未写入log的同时即向Leader发送ACK的反馈,所以对于某一条已经算作commit的消息来讲,在某一时刻,其存在于Leader的log中,以及Replica的内存中。这可以算作一个危险的情况(听起来吓人),因为如果此时集群挂了这条消息就算丢失了,但结合producer的属性(request.required.acks=2 当所有follower都收到消息后返回ack)可以保证在绝大多数情况下消息的安全性。当消息算作commit的时候才会暴露给consumer,并保证at-least-once的投递原则。

2)服务的稳定容错性

前面提到过,Kafka天然支持HA,整个leader/follower机制通过zookeeper调度,它在所有broker中选出一个controller,所有Partition的Leader选举都由controller决定,同时controller也负责增删Topic以及Replica的重新分配。如果Leader挂了,集群将在ISR(in-sync replicas)中选出新的Leader,选举基本原则是:新的Leader必须拥有原来的Leader commit过的所有消息。假如所有的follower都挂了,Kafka会选择第一个“活”过来的Replica(不一定是ISR中的)作为Leader,因为如果此时等待ISR中的Replica是有风险的,假如所有的ISR都无法“活”,那此partition将会变成不可用。

3) 吞吐量

Leader节点负责某一topic(可以分成多个partition)的某一partition的消息的读写,任何发布到此partition的消息都会被直接追加到log文件的尾部,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),同时通过合理的partition,消息可以均匀的分布在不同的partition里面。Kafka基于时间或者partition的大小来删除消息,同时broker是无状态的,consumer的消费状态(offset)是由consumer自己控制的(每一个consumer实例只会消费某一个或多个特定partition的数据,而某个partition的数据只会被某一个特定的consumer实例所消费),也不需要broker通过锁机制去控制消息的消费,所以吞吐量惊人,这也是Kafka吸引人的地方。

最后说下由于zookeeper引起的脑裂(Split Brain)问题:每个consumer分别单独通过Zookeeper判断哪些partition down了,那么不同consumer从Zookeeper“看”到的view就可能不一样,这就会造成错误的reblance尝试。而且有可能所有的consumer都认为rebalance已经完成了,但实际上可能并非如此。