# **Kafka 详解**

### **1 .1 Kafka的特性**

* ****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
* ****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

### **1.2 Kafka一些重要设计思想**

下面介绍先大体介绍一下Kafka的主要设计思想，可以让相关人员在短时间内了解到kafka相关特性，如果想深入研究，后面会对其中每一个特性都做详细介绍。

* ****Consumergroup****：各个consumer可以组成一个组，每个消息只能被组中的一个consumer消费，如果一个消息可以被多个consumer消费的话，那么这些consumer必须在不同的组。
* ****消息状态****：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。
* ****消息持久化****：Kafka中会把消息持久化到本地文件系统中，并且保持极高的效率。
* ****消息有效期****：Kafka会长久保留其中的消息，以便consumer可以多次消费，当然其中很多细节是可配置的。
* ****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，Producer可以决定把消息发到哪个分区，在一个分区中消息的顺序就是Producer发送消息的顺序，一个主题中可以有多个分区，具体分区的数量是可配置的。分区的意义很重大，后面的内容会逐渐体现。
* ****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。
* ****插件支持****：现在不少活跃的社区已经开发出不少插件来拓展Kafka的功能，如用来配合Storm、Hadoop、flume相关的插件。

### **1.3 kafka 应用场景**

* ****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。
* ****消息系统****：解耦和生产者和消费者、缓存消息等。
* ****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。
* ****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。
* ****流式处理****：比如spark streaming和storm
* ****事件源****

### **1.4 Kafka架构组件**

Kafka中发布订阅的对象是topic。我们可以为每类数据创建一个topic，把向topic发布消息的客户端称作producer，从topic订阅消息的客户端称作consumer。Producers和consumers可以同时从多个topic读写数据。一个kafka集群由一个或多个broker服务器组成，它负责持久化和备份具体的kafka消息。

* topic：消息存放的目录即主题
* Producer：生产消息到topic的一方
* Consumer：订阅topic消费消息的一方
* Broker：Kafka的服务实例就是一个broker



### **1.5 Kafka Topic&Partition**

消息发送时都被发送到一个topic，其本质就是一个目录，而topic由是由一些Partition Logs(分区日志)组成,其组织结构如下图所示：



我们可以看到，每个Partition中的消息都是有序的，生产的消息被不断追加到Partition log上，其中的每一个消息都被赋予了一个唯一的offset值。   
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。比如我们设置消息过期时间为2天，那么这2天内的所有消息都会被保存到集群中，数据只有超过了两天才会被清除。   
Kafka需要维持的元数据只有一个–消费消息在Partition中的offset值，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。   
把消息日志以Partition的形式存放有多重考虑，第一，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，因此整个集群就可以适应任意大小的数据了；第二就是可以提高并发，因为可以以Partition为单位读写了。

## **二、Kafka 核心组件**

### **2.1 Replications、Partitions 和Leaders**

通过上面介绍的我们可以知道，kafka中的数据是持久化的并且能够容错的。Kafka允许用户为每个topic设置副本数量，副本数量决定了有几个broker来存放写入的数据。如果你的副本数量设置为3，那么一份数据就会被存放在3台不同的机器上，那么就允许有2个机器失败。一般推荐副本数量至少为2，这样就可以保证增减、重启机器时不会影响到数据消费。如果对数据持久化有更高的要求，可以把副本数量设置为3或者更多。   
Kafka中的topic是以partition的形式存放的，每一个topic都可以设置它的partition数量，Partition的数量决定了组成topic的log的数量。Producer在生产数据时，会按照一定规则（这个规则是可以自定义的）把消息发布到topic的各个partition中。上面将的副本都是以partition为单位的，不过只有一个partition的副本会被选举成leader作为读写用。   
关于如何设置partition值需要考虑的因素。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），因此，如果设置的partition的数量小于consumer的数量，就会有消费者消费不到数据。所以，推荐partition的数量一定要大于同时运行的consumer的数量。另外一方面，建议partition的数量大于集群broker的数量，这样leader partition就可以均匀的分布在各个broker中，最终使得集群负载均衡。在Cloudera,每个topic都有上百个partition。需要注意的是，kafka需要为每个partition分配一些内存来缓存消息数据，如果partition数量越大，就要为kafka分配更大的heap space。

### **2.2 Producers**

Producers直接发送消息到broker上的leader partition，不需要经过任何中介一系列的路由转发。为了实现这个特性，kafka集群中的每个broker都可以响应producer的请求，并返回topic的一些元信息，这些元信息包括哪些机器是存活的，topic的leader partition都在哪，现阶段哪些leader partition是可以直接被访问的。   
Producer客户端自己控制着消息被推送到哪些partition。实现的方式可以是随机分配、实现一类随机负载均衡算法，或者指定一些分区算法。Kafka提供了接口供用户实现自定义的分区，用户可以为每个消息指定一个partitionKey，通过这个key来实现一些hash分区算法。比如，把userid作为partitionkey的话，相同userid的消息将会被推送到同一个分区。   
以Batch的方式推送数据可以极大的提高处理效率，kafka Producer 可以将消息在内存中累计到一定数量后作为一个batch发送请求。Batch的数量大小可以通过Producer的参数控制，参数值可以设置为累计的消息的数量（如500条）、累计的时间间隔（如100ms）或者累计的数据大小(64KB)。通过增加batch的大小，可以减少网络请求和磁盘IO的次数，当然具体参数设置需要在效率和时效性方面做一个权衡。   
Producers可以异步的并行的向kafka发送消息，但是通常producer在发送完消息之后会得到一个future响应，返回的是offset值或者发送过程中遇到的错误。这其中有个非常重要的参数“acks”,这个参数决定了producer要求leader partition 收到确认的副本个数，如果acks设置数量为0，表示producer不会等待broker的响应，所以，producer无法知道消息是否发送成功，这样有可能会导致数据丢失，但同时，acks值为0会得到最大的系统吞吐量。   
若acks设置为1，表示producer会在leader partition收到消息时得到broker的一个确认，这样会有更好的可靠性，因为客户端会等待直到broker确认收到消息。若设置为-1，producer会在所有备份的partition收到消息时得到broker的确认，这个设置可以得到最高的可靠性保证。   
Kafka 消息有一个定长的header和变长的字节数组组成。因为kafka消息支持字节数组，也就使得kafka可以支持任何用户自定义的序列号格式或者其它已有的格式如Apache Avro、protobuf等。Kafka没有限定单个消息的大小，但我们推荐消息大小不要超过1MB,通常一般消息大小都在1~10kB之前。

### **2.3 Consumers**

Kafka提供了两套consumer api，分为high-level api和sample-api。Sample-api 是一个底层的API，它维持了一个和单一broker的连接，并且这个API是完全无状态的，每次请求都需要指定offset值，因此，这套API也是最灵活的。   
在kafka中，当前读到消息的offset值是由consumer来维护的，因此，consumer可以自己决定如何读取kafka中的数据。比如，consumer可以通过重设offset值来重新消费已消费过的数据。不管有没有被消费，kafka会保存数据一段时间，这个时间周期是可配置的，只有到了过期时间，kafka才会删除这些数据。   
High-level API封装了对集群中一系列broker的访问，可以透明的消费一个topic。它自己维持了已消费消息的状态，即每次消费的都是下一个消息。   
High-level API还支持以组的形式消费topic，如果consumers有同一个组名，那么kafka就相当于一个队列消息服务，而各个consumer均衡的消费相应partition中的数据。若consumers有不同的组名，那么此时kafka就相当与一个广播服务，会把topic中的所有消息广播到每个consumer。   


## **三、Kafka核心特性**

### **3.1 压缩**

我们上面已经知道了Kafka支持以集合（batch）为单位发送消息，在此基础上，Kafka还支持对消息集合进行压缩，Producer端可以通过GZIP或Snappy格式对消息集合进行压缩。Producer端进行压缩之后，在Consumer端需进行解压。压缩的好处就是减少传输的数据量，减轻对网络传输的压力，在对大数据处理上，瓶颈往往体现在网络上而不是CPU（压缩和解压会耗掉部分CPU资源）。   
那么如何区分消息是压缩的还是未压缩的呢，Kafka在消息头部添加了一个描述压缩属性字节，这个字节的后两位表示消息的压缩采用的编码，如果后两位为0，则表示消息未被压缩。

### **3.2消息可靠性**

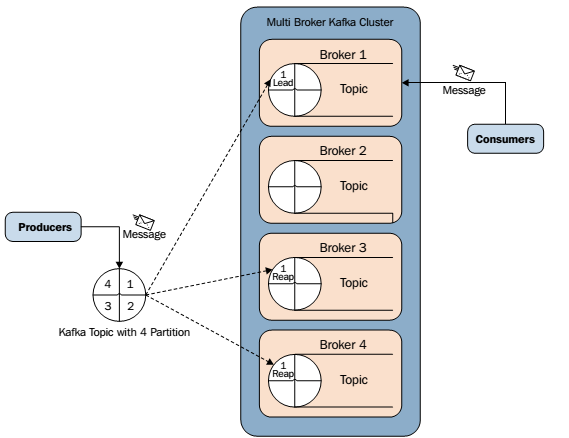
在消息系统中，保证消息在生产和消费过程中的可靠性是十分重要的，在实际消息传递过程中，可能会出现如下三中情况：

* 一个消息发送失败
* 一个消息被发送多次
* 最理想的情况：exactly-once ,一个消息发送成功且仅发送了一次

有许多系统声称它们实现了exactly-once，但是它们其实忽略了生产者或消费者在生产和消费过程中有可能失败的情况。比如虽然一个Producer成功发送一个消息，但是消息在发送途中丢失，或者成功发送到broker，也被consumer成功取走，但是这个consumer在处理取过来的消息时失败了。   
从Producer端看：Kafka是这么处理的，当一个消息被发送后，Producer会等待broker成功接收到消息的反馈（可通过参数控制等待时间），如果消息在途中丢失或是其中一个broker挂掉，Producer会重新发送（我们知道Kafka有备份机制，可以通过参数控制是否等待所有备份节点都收到消息）。   
从Consumer端看：前面讲到过partition，broker端记录了partition中的一个offset值，这个值指向Consumer下一个即将消费message。当Consumer收到了消息，但却在处理过程中挂掉，此时Consumer可以通过这个offset值重新找到上一个消息再进行处理。Consumer还有权限控制这个offset值，对持久化到broker端的消息做任意处理。

### **3.3 备份机制**

备份机制是Kafka0.8版本的新特性，备份机制的出现大大提高了Kafka集群的可靠性、稳定性。有了备份机制后，Kafka允许集群中的节点挂掉后而不影响整个集群工作。一个备份数量为n的集群允许n-1个节点失败。在所有备份节点中，有一个节点作为lead节点，这个节点保存了其它备份节点列表，并维持各个备份间的状体同步。下面这幅图解释了Kafka的备份机制:



### **3.4 Kafka高效性相关设计**

#### **3.4.1 消息的持久化**

Kafka高度依赖文件系统来存储和缓存消息，一般的人认为磁盘是缓慢的，这导致人们对持久化结构具有竞争性持怀疑态度。其实，磁盘远比你想象的要快或者慢，这决定于我们如何使用磁盘。   
一个和磁盘性能有关的关键事实是：磁盘驱动器的吞吐量跟寻到延迟是相背离的，也就是所，线性写的速度远远大于随机写。比如：在一个6 7200rpm SATA RAID-5 的磁盘阵列上线性写的速度大概是600M/秒，但是随机写的速度只有100K/秒，两者相差将近6000倍。线性读写在大多数应用场景下是可以预测的，因此，操作系统利用read-ahead和write-behind技术来从大的数据块中预取数据，或者将多个逻辑上的写操作组合成一个大写物理写操作中。更多的讨论可以在[ACMQueueArtical](http://queue.acm.org/detail.cfm?id=1563874" \t "http://blog.csdn.net/suifeng3051/article/details/_blank)中找到，他们发现，对磁盘的线性读在有些情况下可以比内存的随机访问要快一些。   
为了补偿这个性能上的分歧，现代操作系统都会把空闲的内存用作磁盘缓存，尽管在内存回收的时候会有一点性能上的代价。所有的磁盘读写操作会在这个统一的缓存上进行。   
此外，如果我们是在JVM的基础上构建的，熟悉java内存应用管理的人应该清楚以下两件事情：

1. 一个对象的内存消耗是非常高的，经常是所存数据的两倍或者更多。
2. 随着堆内数据的增多，Java的垃圾回收会变得非常昂贵。

基于这些事实，利用文件系统并且依靠页缓存比维护一个内存缓存或者其他结构要好——我们至少要使得可用的缓存加倍，通过自动访问可用内存，并且通过存储更紧凑的字节结构而不是一个对象，这将有可能再次加倍。这么做的结果就是在一台32GB的机器上，如果不考虑GC惩罚，将最多有28-30GB的缓存。此外，这些缓存将会一直存在即使服务重启，然而进程内缓存需要在内存中重构（10GB缓存需要花费10分钟）或者它需要一个完全冷缓存启动（非常差的初始化性能）。它同时也简化了代码，因为现在所有的维护缓存和文件系统之间内聚的逻辑都在操作系统内部了，这使得这样做比one-off in-process attempts更加高效与准确。如果你的磁盘应用更加倾向于顺序读取，那么read-ahead在每次磁盘读取中实际上获取到这人缓存中的有用数据。   
以上这些建议了一个简单的设计：不同于维护尽可能多的内存缓存并且在需要的时候刷新到文件系统中，我们换一种思路。所有的数据不需要调用刷新程序，而是立刻将它写到一个持久化的日志中。事实上，这仅仅意味着，数据将被传输到内核页缓存中并稍后被刷新。我们可以增加一个配置项以让系统的用户来控制数据在什么时候被刷新到物理硬盘上。

#### **3.4.2 常数时间性能保证**

消息系统中持久化数据结构的设计通常是维护者一个和消费队列有关的B树或者其它能够随机存取结构的元数据信息。B树是一个很好的结构，可以用在事务型与非事务型的语义中。但是它需要一个很高的花费，尽管B树的操作需要O(logN)。通常情况下，这被认为与常数时间等价，但这对磁盘操作来说是不对的。磁盘寻道一次需要10ms，并且一次只能寻一个，因此并行化是受限的。   
直觉上来讲，一个持久化的队列可以构建在对一个文件的读和追加上，就像一般情况下的日志解决方案。尽管和B树相比，这种结构不能支持丰富的语义，但是它有一个优点，所有的操作都是常数时间，并且读写之间不会相互阻塞。这种设计具有极大的性能优势：最终系统性能和数据大小完全无关，服务器可以充分利用廉价的硬盘来提供高效的消息服务。   
事实上还有一点，磁盘空间的无限增大而不影响性能这点，意味着我们可以提供一般消息系统无法提供的特性。比如说，消息被消费后不是立马被删除，我们可以将这些消息保留一段相对比较长的时间（比如一个星期）。

#### **3.4.3 进一步提高效率**

我们已经为效率做了非常多的努力。但是有一种非常主要的应用场景是：处理Web活动数据，它的特点是数据量非常大，每一次的网页浏览都会产生大量的写操作。更进一步，我们假设每一个被发布的消息都会被至少一个consumer消费，因此我们更要怒路让消费变得更廉价。   
通过上面的介绍，我们已经解决了磁盘方面的效率问题，除此之外，在此类系统中还有两类比较低效的场景：

* 太多小的I/O操作
* 过多的字节拷贝

为了减少大量小I/O操作的问题，kafka的协议是围绕消息集合构建的。Producer一次网络请求可以发送一个消息集合，而不是每一次只发一条消息。在server端是以消息块的形式追加消息到log中的，consumer在查询的时候也是一次查询大量的线性数据块。消息集合即MessageSet，实现本身是一个非常简单的API，它将一个字节数组或者文件进行打包。所以对消息的处理，这里没有分开的序列化和反序列化的上步骤，消息的字段可以按需反序列化（如果没有需要，可以不用反序列化）。   
另一个影响效率的问题就是字节拷贝。为了解决字节拷贝的问题，kafka设计了一种“标准字节消息”，Producer、Broker、Consumer共享这一种消息格式。Kakfa的message log在broker端就是一些目录文件，这些日志文件都是MessageSet按照这种“标准字节消息”格式写入到磁盘的。   
维持这种通用的格式对这些操作的优化尤为重要：持久化log 块的网络传输。流行的unix操作系统提供了一种非常高效的途径来实现页面缓存和socket之间的数据传递。在Linux操作系统中，这种方式被称作：sendfile system call（Java提供了访问这个系统调用的方法：FileChannel.transferTo api）。

为了理解sendfile的影响，需要理解一般的将数据从文件传到socket的路径：

1. 操作系统将数据从磁盘读到内核空间的页缓存中
2. 应用将数据从内核空间读到用户空间的缓存中
3. 应用将数据写回内核空间的socket缓存中
4. 操作系统将数据从socket缓存写到网卡缓存中，以便将数据经网络发出

这种操作方式明显是非常低效的，这里有四次拷贝，两次系统调用。如果使用sendfile，就可以避免两次拷贝：操作系统将数据直接从页缓存发送到网络上。所以在这个优化的路径中，只有最后一步将数据拷贝到网卡缓存中是需要的。   
我们期望一个主题上有多个消费者是一种常见的应用场景。利用上述的zero-copy，数据只被拷贝到页缓存一次，然后就可以在每次消费时被重得利用，而不需要将数据存在内存中，然后在每次读的时候拷贝到内核空间中。这使得消息消费速度可以达到网络连接的速度。这样以来，通过页面缓存和sendfile的结合使用，整个kafka集群几乎都已以缓存的方式提供服务，而且即使下游的consumer很多，也不会对整个集群服务造成压力。   
关于sendfile和zero-copy，请参考：[zero-copy](https://www.ibm.com/developerworks/linux/library/j-zerocopy/" \t "http://blog.csdn.net/suifeng3051/article/details/_blank)

## **四、Kafka集群部署**

### **4.1 集群部署**

为了提高性能，推荐采用专用的服务器来部署kafka集群，尽量与hadoop集群分开，因为kafka依赖磁盘读写和大的页面缓存，如果和hadoop共享节点的话会影响其使用页面缓存的性能。   
Kafka集群的大小需要根据硬件的配置、生产者消费者的并发数量、数据的副本个数、数据的保存时长综合确定。   
磁盘的吞吐量尤为重要，因为通常kafka的瓶颈就在磁盘上。   
Kafka依赖于zookeeper，建议采用专用服务器来部署zookeeper集群，zookeeper集群的节点采用偶数个，一般建议用3、5、7个。注意zookeeper集群越大其读写性能越慢，因为zookeeper需要在节点之间同步数据。一个3节点的zookeeper集群允许一个节点失败，一个5节点集群允许2个几点失败。

### **4.2 集群大小**

有很多因素决定着kafka集群需要具备存储能力的大小，最准确的衡量办法就是模拟负载来测算一下，Kafka本身也提供了负载测试的工具。   
如果不想通过模拟实验来评估集群大小，最好的办法就是根据硬盘的空间需求来推算。下面我就根据网络和磁盘吞吐量需求来做一下估算。   
我们做如下假设：

* W：每秒写多少MB
* R ：副本数
* C ：Consumer的数量

一般的来说，kafka集群瓶颈在于网络和磁盘吞吐量，所以我们先评估一下集群的网络和磁盘需求。   
对于每条消息，每个副本都要写一遍，所以整体写的速度是W\*R。读数据的部分主要是集群内部各个副本从leader同步消息读和集群外部的consumer读，所以集群内部读的速率是(R-1)\*W,同时，外部consumer读的速度是C\*W，因此：

* Write：W\*R
* Read：（R-1）\*W+C\*W

需要注意的是，我们可以在读的时候缓存部分数据来减少IO操作，如果一个集群有M MB内存，写的速度是W MB/sec，则允许M/(W\*R) 秒的写可以被缓存。如果集群有32GB内存，写的速度是50MB/s的话，则可以至少缓存10分钟的数据。