开源代码研究报告

姓名：陈镜宇

学号：2017104137

项目位置：https://github.com/apache/kafka

背景介绍

**Kafka**简介

Kafka是一种分布式的，基于发布/订阅的消息系统。主要设计目标如下：

* 以时间复杂度为O(1)的方式提供消息持久化能力，并保证即使对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，及分布式消息消费，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

产生背景

Kafka是一个消息系统，用作LinkedIn的活动流（Activity Stream）和运营数据处理管道（Pipeline）的基础。活动流数据是几乎所有站点在对其网站使用情况做报表时都要用到的数据中最常规的部分。活动数据包括页面访问量（Page View）、被查看内容方面的信息以及搜索情况等内容。这种数据通常的处理方式是先把各种活动以日志的形式写入某种文件，然后周期性地对这些文件进行统计分析。运营数据指的3是服务器的性能数据（CPU、IO使用率、请求时间、服务日志等等数据)。运营数据的统计方法种类繁多。

基本架构图



为什么要用**Message Queue**

* 解耦   
  在项目启动之初来预测将来项目会碰到什么需求，是极其困难的。消息队列在处理过程中间插入了一个隐含的、基于数据的接口层，两边的处理过程都要实现这一接口。这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
* 冗余   
  有时在处理数据的时候处理过程会失败。除非数据被持久化，否则将永远丢失。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。在被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，需要你的处理过程明确的指出该消息已经被处理完毕，确保你的 数据被安全的保存直到你使用完毕。
* 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。不需要改变代码、不需要调节参数。扩展就像调大电力按钮一样简单。
* 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增长的访问压力，而不是因为超出负荷的请求而完全崩溃。
* 可恢复性   
  当体系的一部分组件失效，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。而这种允许重试或者延后处理请求的能力通常是造就一个略感不便的用户和一个沮丧透顶的用户之间的区别。
* 送达保证   
  消息队列提供的冗余机制保证了消息能被实际的处理，只要一个进程读取了该队列即可。在此基础上，IronMQ提供了一个”只送达一次”保证。无论有多少进 程在从队列中领取数据，每一个消息只能被处理一次。这之所以成为可能，是因为获取一个消息只是”预定”了这个消息，暂时把它移出了队列。除非客户端明确的 表示已经处理完了这个消息，否则这个消息会被放回队列中去，在一段可配置的时间之后可再次被处理。
* 顺序保证   
  在许多情况下，数据处理的顺序都很重要。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。IronMO保证消息浆糊通过FIFO（先进先出）的顺序来处理，因此消息在队列中的位置就是从队列中检索他们的位置。
* 缓冲   
  在任何重要的系统中，都会有需要不同的处理时间的元素。例如,加载一张图片比应用过滤器花费更少的时间。消息队列通过一个缓冲层来帮助任务最高效率的执行—写入队列的处理会尽可能的快速，而不受从队列读的预备处理的约束。该缓冲有助于控制和优化数据流经过系统的速度。
* 理解数据流   
  在一个分布式系统里，要得到一个关于用户操作会用多长时间及其原因的总体印象，是个巨大的挑战。消息系列通过消息被处理的频率，来方便的辅助确定那些表现不佳的处理过程或领域，这些地方的数据流都不够优化。
* 异步通信   
  很多时候，你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，允许你把一个消息放入队列，但并不立即处理它。你想向队列中放入多少消息就放多少，然后在你乐意的时候再去处理它们。

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down机，数据将会丢失。其中，Twitter的Storm中默认使用ZeroMQ作为数据流的传 输。
* **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Publish/Subscribe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，自动实现复杂均衡；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，这一点也是本课题所研究系统所看重的。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**解析

**Terminology**

* **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consuer 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**架构

一个典型的kafka集群中包含若干producer（可以是web前端产生的page view，或者是服务器日志，系统CPU、memory等），若干broker（Kafka支持水平扩展，一般broker数量越多，集群吞吐率越高）， 若干consumer group，以及一个 [Zookeeper](http://zookeeper.apache.org/) 集群。Kafka通过Zookeeper管理集群配置，选举leader，以及在consumer group发生变化时进行rebalance。producer使用push模式将消息发布到broker，consumer使用pull模式从 broker订阅并消费消息。

**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](https://github.com/facebookarchive/scribe) 和Cloudera的 [Flume](http://flume.apache.org/) ,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。

**Topic & Partition**

Topic在逻辑上可以被认为是一个在的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。

每个日志文件都是“log entries”序列，每一个 log entry 包含一个4字节整型数（值为N），其后跟N个字节的消息体。每条消息都有一个当前partition下唯一的64字节的offset，它指明了这条消息的起始位置。磁盘上存储的消费格式如下：

message length ： 4 bytes (value: 1+4+n)

“magic” value ： 1 byte

crc ： 4 bytes

payload ： n bytes

这个“log entries”并非由一个文件构成，而是分成多个segment，每个segment名为该segment第一条消息的offset和“.kafka”组成。另外会有一个索引文件，它标明了每个segment下包含的 log entry 的offset范围，如下图所示。

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(如下所示)，当然也可以在topic创建之后去修改parition数量。

# The default number of log partitions per topic. More partitions allow greater

# parallelism for consumption, but this will also result in more files across

# the brokers.

num.partitions=3

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。 paritition机制可以通过指定producer的paritition. class这一参数来指定，该class必须实现 kafka.producer.Partitioner 接口。本例中如果key可以被解析为整数则将对应的整数与partition总数取余，该消息会被发送到该数对应的partition。（每个parition都会有个序号）

import kafka.producer.Partitioner;

import kafka.utils.VerifiableProperties;

public class JasonPartitioner<T> implements Partitioner {

public JasonPartitioner(VerifiableProperties verifiableProperties) {}

@Override

public int partition(Object key, int numPartitions) {

try {

int partitionNum = Integer.parseInt((String) key);

return Math.abs(Integer.parseInt((String) key) % numPartitions);

} catch (Exception e) {

return Math.abs(key.hashCode() % numPartitions);

}

}

}

如果将上例中的class作为partition.class，并通过如下代码发送20条消息（key分别为0，1，2，3）至topic2（包含4个partition）。

public void sendMessage() throws InterruptedException{

　　for(int i = 1; i <= 5; i++){

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

　　 for(int j = 0; j < 4; j++）{

　　 messageList.add(new KeyedMessage<String, String>("topic2", j+"", "The " + i + " message for key " + j));

　　 }

　　 producer.send(messageList);

}

　　producer.close();

}

则key相同的消息会被发送并存储到同一个partition里，而且key的序号正好和partition序号相同。（partition序号从0开始，本例中的key也正好从0开始）。如下图所示。

对于传统的message queue而言，一般会删除已经被消费的消息，而Kafka集群会保留所有的消息，无论其被消费与否。当然，因为磁盘限制，不可能永久保留所有数据（实际 上也没必要），因此Kafka提供两种策略去删除旧数据。一是基于时间，二是基于partition文件大小。例如可以通过配置 $KAFKA\_HOME/config/server.properties ，让Kafka删除一周前的数据，也可通过配置让Kafka在partition文件超过1GB时删除旧数据，如下所示。

　　############################# Log Retention Policy #############################

# The following configurations control the disposal of log segments. The policy can

# be set to delete segments after a period of time, or after a given size has accumulated.

# A segment will be deleted whenever \*either\* of these criteria are met. Deletion always happens

# from the end of the log.

# The minimum age of a log file to be eligible for deletion

log.retention.hours=168

# A size-based retention policy for logs. Segments are pruned from the log as long as the remaining

# segments don't drop below log.retention.bytes.

#log.retention.bytes=1073741824

# The maximum size of a log segment file. When this size is reached a new log segment will be created.

log.segment.bytes=1073741824

# The interval at which log segments are checked to see if they can be deleted according

# to the retention policies

log.retention.check.interval.ms=300000

# By default the log cleaner is disabled and the log retention policy will default to

#just delete segments after their retention expires.

# If log.cleaner.enable=true is set the cleaner will be enabled and individual logs

#can then be marked for log compaction.

log.cleaner.enable=false

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，所以这里删除文件与Kafka性能无关，选择怎样的删除策略 只与磁盘以及具体的需求有关。另外，Kafka会为每一个consumer group保留一些metadata信息—当前消费的消息的position，也即offset。这个offset由consumer控制。正常情况下 consumer会在消费完一条消息后线性增加这个offset。当然，consumer也可将offset设成一个较小的值，重新消费一些消息。因为 offet由consumer控制，所以Kafka broker是无状态的，它不需要标记哪些消息被哪些consumer过，不需要通过broker去保证同一个consumer group只有一个consumer能消费某一条消息，因此也就不需要锁机制，这也为Kafka的高吞吐率提供了有力保障。

**Replication & Leader election**

Kafka从0.8开始提供partition级别的replication，replication的数量可在 $KAFKA\_HOME/config/server.properties 中配置。

该 Replication与leader election配合提供了自动的failover机制。replication对Kafka的吞吐率是有一定影响的，但极大的增强了可用性。默认情况 下，Kafka的replication数量为1。 每个partition都有一个唯一的leader，所有的读写操作都在leader上完成，leader批量从leader上pull数据。一般情况下 partition的数量大于等于broker的数量，并且所有partition的leader均匀分布在broker上。follower上的日志和 其leader上的完全一样。

和大部分分布式系统一样，Kakfa处理失败需要明确定义一个broker是否alive。对于Kafka而言，Kafka存活包含两个条件， 一是它必须维护与Zookeeper的session(这个通过Zookeeper的heartbeat机制来实现)。二是follower必须能够及时 将leader的writing复制过来，不能“落后太多”。

leader会track“in sync”的node list。如果一个follower宕机，或者落后太多，leader将把它从”in sync” list中移除。这里所描述的“落后太多”指follower复制的消息落后于leader后的条数超过预定值，该值可在 $KAFKA\_HOME/config/server.properties 中配置

#If a replica falls more than this many messages behind the leader, the leader will remove the follower from ISR and treat it as dead

replica.lag.max.messages=4000

#If a follower hasn't sent any fetch requests for this window of time, the leader will remove the follower from ISR (in-sync replicas) and treat it as dead

replica.lag.time.max.ms=10000

需要说明的是，Kafka只解决”fail/recover”，不处理“Byzantine”（“拜占庭”）问题。

一条消息只有被“in sync” list里的所有follower都从leader复制过去才会被认为已提交。这样就避免了部分数据被写进了leader，还没来得及被任何 follower复制就宕机了，而造成数据丢失（consumer无法消费这些数据）。而对于producer而言，它可以选择是否等待消息 commit，这可以通过 request.required.acks来设置。这种机制确保了只要“in sync” list有一个或以上的flollower，一条被commit的消息就不会丢失。

这里的复制机制即不是同步复制，也不是单纯的异步复制。事实上，同步复制要求“活着的”follower都复制完，这条消息才会被认为 commit，这种复制方式极大的影响了吞吐率（高吞吐率是Kafka非常重要的一个特性）。而异步复制方式下，follower异步的从leader复 制数据，数据只要被leader写入log就被认为已经commit，这种情况下如果follwer都落后于leader，而leader突然宕机，则会 丢失数据。而Kafka的这种使用“in sync” list的方式则很好的均衡了确保数据不丢失以及吞吐率。follower可以批量的从leader复制数据，这样极大的提高复制性能（批量写磁盘），极 大减少了follower与leader的差距（前文有说到，只要follower落后leader不太远，则被认为在“in sync” list里）。

上文说明了Kafka是如何做replication的，另外一个很重要的问题是当leader宕机了，怎样在follower中选举出新的 leader。因为follower可能落后许多或者crash了，所以必须确保选择“最新”的follower作为新的leader。一个基本的原则就 是，如果leader不在了，新的leader必须拥有原来的leader commit的所有消息。这就需要作一个折衷，如果leader在标明一条消息被commit前等待更多的follower确认，那在它die之后就有更 多的follower可以作为新的leader，但这也会造成吞吐率的下降。

一种非常常用的选举leader的方式是“majority 灵秀”（“少数服从多数”），但Kafka并未采用这种方式。这种模式下，如果我们有2f+1个replica（包含leader和follower）， 那在commit之前必须保证有f+1个replica复制完消息，为了保证正确选出新的leader，fail的replica不能超过f个。因为在剩 下的任意f+1个replica里，至少有一个replica包含有最新的所有消息。这种方式有个很大的优势，系统的latency只取决于最快的几台 server，也就是说，如果replication factor是3，那latency就取决于最快的那个follower而非最慢那个。majority vote也有一些劣势，为了保证leader election的正常进行，它所能容忍的fail的follower个数比较少。如果要容忍1个follower挂掉，必须要有3个以上的 replica，如果要容忍2个follower挂掉，必须要有5个以上的replica。也就是说，在生产环境下为了保证较高的容错程度，必须要有大量 的replica，而大量的replica又会在大数据量下导致性能的急剧下降。这就是这种算法更多用在 [Zookeeper](http://zookeeper.apache.org/) 这种共享集群配置的系统中而很少在需要存储大量数据的系统中使用的原因。例如HDFS的HA feature是基于 [majority-vote-based journal](http://blog.cloudera.com/blog/2012/10/quorum-based-journaling-in-cdh4-1) ，但是它的数据存储并没有使用这种expensive的方式。

实际上，leader election算法非常多，比如Zookeper的 [Zab](http://web.stanford.edu/class/cs347/reading/zab.pdf) , [Raft](https://ramcloud.stanford.edu/wiki/download/attachments/11370504/raft.pdf) 和 [Viewstamped Replication](http://pmg.csail.mit.edu/papers/vr-revisited.pdf) 。而Kafka所使用的leader election算法更像微软的 [PacificA](http://research.microsoft.com/apps/pubs/default.aspx?id=66814)算法。

Kafka在Zookeeper中动态维护了一个ISR（in-sync replicas） set，这个set里的所有replica都跟上了leader，只有ISR里的成员才有被选为leader的可能。在这种模式下，对于f+1个 replica，一个Kafka topic能在保证不丢失已经ommit的消息的前提下容忍f个replica的失败。在大多数使用场景中，这种模式是非常有利的。事实上，为了容忍f个 replica的失败，majority vote和ISR在commit前需要等待的replica数量是一样的，但是ISR需要的总的replica的个数几乎是majority vote的一半。

虽然majority vote与ISR相比有不需等待最慢的server这一优势，但是Kafka作者认为Kafka可以通过producer选择是否被commit阻塞来改善这一问题，并且节省下来的replica和磁盘使得ISR模式仍然值得。

上文提到，在ISR中至少有一个follower时，Kafka可以确保已经commit的数据不丢失，但如果某一个partition的所有replica都挂了，就无法保证数据不丢失了。这种情况下有两种可行的方案：

* 等待ISR中的任一个replica“活”过来，并且选它作为leader
* 选择第一个“活”过来的replica（不一定是ISR中的）作为leader

这就需要在可用性和一致性当中作出一个简单的平衡。如果一定要等待ISR中的replica“活”过来，那不可用的时间就可能会相对较长。而且如 果ISR中的所有replica都无法“活”过来了，或者数据都丢失了，这个partition将永远不可用。选择第一个“活”过来的replica作为 leader，而这个replica不是ISR中的replica，那即使它并不保证已经包含了所有已commit的消息，它也会成为leader而作为 consumer的数据源（前文有说明，所有读写都由leader完成）。Kafka0.8.\*使用了第二种方式。根据Kafka的文档，在以后的版本 中，Kafka支持用户通过配置选择这两种方式中的一种，从而根据不同的使用场景选择高可用性还是强一致性。

上文说明了一个parition的replication过程，然尔Kafka集群需要管理成百上千个partition，Kafka通过 round-robin的方式来平衡partition从而避免大量partition集中在了少数几个节点上。同时Kafka也需要平衡leader的 分布，尽可能的让所有partition的leader均匀分布在不同broker上。另一方面，优化leadership election的过程也是很重要的，毕竟这段时间相应的partition处于不可用状态。一种简单的实现是暂停宕机的broker上的所有 partition，并为之选举leader。实际上，Kafka选举一个broker作为controller，这个controller通过 watch Zookeeper检测所有的broker failure，并负责为所有受影响的parition选举leader，再将相应的leader调整命令发送至受影响的broker，过程如下图所示。

这样做的好处是，可以批量的通知leadership的变化，从而使得选举过程成本更低，尤其对大量的partition而言。如果 controller失败了，幸存的所有broker都会尝试在Zookeeper中创建/controller->{this broker id}，如果创建成功（只可能有一个创建成功），则该broker会成为controller，若创建不成功，则该broker会等待新 controller的命令。

**Consumer group**

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）

很多传统的message queue都会在消息被消费完后将消息删除，一方面避免重复消费，另一方面可以保证queue的长度比较少，提高效率。而如上文所将，Kafka并不删除 已消费的消息，为了实现传统message queue消息只被消费一次的语义，Kafka保证保证同一个consumer group里只有一个consumer会消费一条消息。与传统message queue不同的是，Kafka还允许不同consumer group同时消费同一条消息，这一特性可以为消息的多元化处理提供了支持。实际上，Kafka的设计理念之一就是同时提供离线处理和实时处理。根据这一 特性，可以使用Storm这种实时流处理系统对消息进行实时在线处理，同时使用Hadoop这种批处理系统进行离线处理，还可以同时将数据实时备份到另一 个数据中心，只需要保证这三个操作所使用的consumer在不同的consumer group即可。下图展示了Kafka在Linkedin的一种简化部署。

为了更清晰展示Kafka consumer group的特性，笔者作了一项测试。创建一个topic (名为topic1)，创建一个属于group1的consumer实例，并创建三个属于group2的consumer实例，然后通过producer 向topic1发送key分别为1，2，3r的消息。结果发现属于group1的consumer收到了所有的这三条消息，同时group2中的3个 consumer分别收到了key为1，2，3的消息。如下图所示。

**Consumer Rebalance**

Kafka保证同一consumer group中只有一个consumer会消息某条消息，实际上，Kafka保证的是稳定状态下每一个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。这样设计的劣势是无法让同一个 consumer group里的consumer均匀消费数据，优势是每个consumer不用都跟大量的broker通信，减少通信开销，同时也降低了分配难度，实现也 更简单。另外，因为同一个partition里的数据是有序的，这种设计可以保证每个partition里的数据也是有序被消费。

如果某consumer group中consumer数量少于partition数量，则至少有一个consumer会消费多个partition的数据，如果consumer 的数量与partition数量相同，则正好一个consumer消费一个partition的数据，而如果consumer的数量多于 partition的数量时，会有部分consumer无法消费该topic下任何一条消息。

如下例所示，如果topic1有0，1，2共三个partition，当group1只有一个consumer(名为consumer1)时，该 consumer可消费这3个partition的所有数据。

增加一个consumer(consumer2)后，其中一个consumer（consumer1）可消费2个partition的数据，另外一个consumer(consumer2)可消费另外一个partition的数据。

再增加一个consumer(consumer3)后，每个consumer可消费一个partition的数据。consumer1消费partition0，consumer2消费partition1，consumer3消费partition2

再增加一个consumer（consumer4）后，其中3个consumer可分别消费一个partition的数据，另外一个consumer（consumer4）不能消费topic1任何数据。

此时关闭consumer1，剩下的consumer可分别消费一个partition的数据。

接着关闭consumer2，剩下的consumer3可消费2个partition，consumer4可消费1个partition。

再关闭consumer3，剩下的consumer4可同时消费topic1的3个partition。

consumer rebalance算法如下：

* Sort PT (all partitions in topic T)
* Sort CG(all consumers in consumer group G)
* Let i be the index position of Ci in CG and let N=size(PT)/size(CG)
* Remove current entries owned by Ci from the partition owner registry
* Assign partitions from i *N to (i+1)* N-1 to consumer Ci
* Add newly assigned partitions to the partition owner registry

目前consumer rebalance的控制策略是由每一个consumer通过Zookeeper完成的。具体的控制方式如下：

* Register itself in the consumer id registry under its group.
* Register a watch on changes under the consumer id registry.
* Register a watch on changes under the broker id registry.
* If the consumer creates a message stream using a topic filter, it also registers a watch on changes under the broker topic registry.
* Force itself to rebalance within in its consumer group.  
  在 这种策略下，每一个consumer或者broker的增加或者减少都会触发consumer rebalance。因为每个consumer只负责调整自己所消费的partition，为了保证整个consumer group的一致性，所以当一个consumer触发了rebalance时，该consumer group内的其它所有consumer也应该同时触发rebalance。

目前（2015-01-19）最新版（0.8.2）Kafka采用的是上述方式。但该方式有不利的方面：

* **Herd effect**   
  任何broker或者consumer的增减都会触发所有的consumer的rebalance
* **Split Brain**   
  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。

根据Kafka官方文档，Kafka作者正在考虑在还未发布的 [0.9.x版本中使用中心协调器(coordinator)](https://cwiki.apache.org/confluence/display/KAFKA/Kafka+0.9+Consumer+Rewrite+Design) 。大体思想是选举出一个broker作为coordinator，由它watch Zookeeper，从而判断是否有partition或者consumer的增减，然后生成rebalance命令，并检查是否这些rebalance 在所有相关的consumer中被执行成功，如果不成功则重试，若成功则认为此次rebalance成功（这个过程跟replication controller非常类似，所以我很奇怪为什么当初设计replication controller时没有使用类似方式来解决consumer rebalance的问题）。流程如下：

消息**Deliver guarantee**

通过上文介绍，想必读者已经明天了producer和consumer是如何工作的，以及Kafka是如何做replication的，接下来要 讨论的是Kafka如何确保消息在producer和consumer之间传输。有这么几种可能的delivery guarantee：

* At most once 消息可能会丢，但绝不会重复传输
* At least one 消息绝不会丢，但可能会重复传输
* Exactly once 每条消息肯定会被传输一次且仅传输一次，很多时候这是用户所想要的。Kafka的delivery guarantee semantic非常直接。当producer向broker发送消息时，一旦这条消息被commit，因数replication的存在，它就不会丢。 但是如果producer发送数据给broker后，遇到的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。这一点 有点像向一个自动生成primary key的数据库表中插入数据。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primary key的东西，发生故障时幂等性的retry多次，这样就做到了 Exactly one 。截止到目前(Kafka 0.8.2版本，2015-01-25)，这一feature还并未实现，有希望在Kafka未来的版本中实现。（所以目前默认情况下一条消息从producer和broker是确保了 At least once ，但可通过设置producer异步发送实现 At most once ）。  
  接下来讨论的是消息从broker到consumer的delivery guarantee semantic。（仅针对Kafka consumer high level API）。consumer在从broker读取消息后，可以选择commit，该操作会在Zookeeper中存下该consumer在该 partition下读取的消息的offset。该consumer下一次再读该partition时会从下一条开始读取。如未commit，下一次读取 的开始位置会跟上一次commit之后的开始位置相同。当然可以将consumer设置为autocommit，即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是确保了 Exactly once 。但实际上实际使用中consumer并非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消息从broker和consumer的delivery guarantee semantic。
* 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
* 读完消息先处理再commit。这种模式下，如果处理完了消息在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 。在很多情况使用场景下，消息都有一个primary key，所以消息的处理往往具有幂等性，即多次处理这一条消息跟只处理一次是等效的，那就可以认为是 Exactly once 。（人个感觉这种说法有些牵强，毕竟它不是Kafka本身提供的机制，而且primary key本身不保证操作的幂等性。而且实际上我们说delivery guarantee semantic是讨论被处理多少次，而非处理结果怎样，因为处理方式多种多样，我们的系统不应该把处理过程的特性—如是否幂等性，当成Kafka本身的 feature）
* 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low level 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 。而 Exactly once 要求与目标存储系统协作，幸运的是Kafka提供的offset可以使用这种方式非常直接非常容易。

**Benchmark**

**producer**吞吐率

该项测试只测producer的吞吐率，也就是数据只被持久化，没有consumer读数据。

**1**个**producer**线程，无**replication**

在这一测试中，创建了一个包含6个partition且没有replication的topic。然后通过一个线程尽可能快的生成50 million条比较短（payload100字节长）的消息。测试结果是 ***821,557 records/second***（ ***78.3MB/second***）。

之所以使用短消息，是因为对于消息系统来说这种使用场景更难。因为如果使用MB/second来表征吞吐率，那发送长消息无疑能使得测试结果更好。

整个测试中，都是用每秒钟delivery的消息的数量乘以payload的长度来计算MB/second的，没有把消息的元信息算在内，所以 实际的网络使用量会比这个大。对于本测试来说，每次还需传输额外的22个字节，包括一个可选的key，消息长度描述，CRC等。另外，还包含一些请求相关 的overhead，比如topic，partition，acknowledgement等。这就导致我们比较难判断是否已经达到网卡极限，但是把这些 overhead都算在吞吐率里面应该更合理一些。因此，我们已经基本达到了网卡的极限。

初步观察此结果会认为它比人们所预期的要高很多，尤其当考虑到Kafka要把数据持久化到磁盘当中。实际上，如果使用随机访问数据系统，比如 RDBMS，或者key-velue store，可预期的最高访问频率大概是5000到50000个请求每秒，这和一个好的RPC层所能接受的远程请求量差不多。而该测试中远超于此的原因有 两个。

* Kafka确保写磁盘的过程是线性磁盘I/O，测试中使用的6块廉价磁盘线性I/O的最大吞吐量是822MB/second，这已经远大于 1Gb网卡所能带来的吞吐量了。许多消息系统把数据持久化到磁盘当成是一个开销很大的事情，这是因为他们对磁盘的操作都不是线性I/O。
* 在每一个阶段，Kafka都尽量使用批量处理。如果想了解批处理在I/O操作中的重要性，可以参考David Patterson的” [Latency Lags Bandwidth](http://www.ll.mit.edu/HPEC/agendas/proc04/invited/patterson_keynote.pdf) “

**1**个**producer**线程，**3**个异步**replication**

该项测试与上一测试基本一样，唯一的区别是每个partition有3个replica（所以网络传输的和写入磁盘的总的数据量增加了3倍）。每一个 broker即要写作为leader的partition，也要读（从leader读数据）写（将数据写到磁盘）作为follower的 partition。测试结果为 ***786,980 records/second***（ ***75.1MB/second***）。

该项测试中replication是异步的，也就是说broker收到数据并写入本地磁盘后就acknowledge producer，而不必等所有replica都完成replication。也就是说，如果leader crash了，可能会丢掉一些最新的还未备份的数据。但这也会让message acknowledgement延迟更少，实时性更好。

这项测试说明，replication可以很快。整个集群的写能力可能会由于3倍的replication而只有原来的三分之一，但是对于每一个producer来说吞吐率依然足够好。

**1**个**producer**线程，**3**个同步**replication**

该项测试与上一测试的唯一区别是replication是同步的，每条消息只有在被 in sync 集合里的所有replica都复制过去后才会被置为committed（此时broker会向producer发送acknowledgement）。在这种模式下，Kafka可以保证即使leader crash了，也不会有数据丢失。测试结果为 ***421,823 records/second***（ ***40.2MB/second***）。

Kafka同步复制与异步复制并没有本质的不同。leader会始终track follower replica从而监控它们是否还alive，只有所有 in sync 集合里的replica都acknowledge的消息才可能被consumer所消费。而对follower的等待影响了吞吐率。可以通过增大batch size来改善这种情况，但为了避免特定的优化而影响测试结果的可比性，本次测试并没有做这种调整。

**3**个**producer,3**个异步**replication**

该测试相当于把上文中的1个producer,复制到了3台不同的机器上（在1台机器上跑多个实例对吞吐率的增加不会有太大帮忙，因为网卡已经基本饱和了），这3个producer同时发送数据。整个集群的吞吐率为 ***2,024,032 records/second***（ ***193,0MB/second***）。

**Producer Throughput Vs. Stored Data**

消息系统的一个潜在的危险是当数据能都存于内存时性能很好，但当数据量太大无法完全存于内存中时（然后很多消息系统都会删除已经被消费的数据， 但当消费速度比生产速度慢时，仍会造成数据的堆积），数据会被转移到磁盘，从而使得吞吐率下降，这又反过来造成系统无法及时接收数据。这样就非常糟糕，而 实际上很多情景下使用queue的目的就是解决数据消费速度和生产速度不一致的问题。

但Kafka不存在这一问题，因为Kafka始终以O（1）的时间复杂度将数据持久化到磁盘，所以其吞吐率不受磁盘上所存储的数据量的影响。为了验证这一特性，做了一个长时间的大数据量的测试，下图是吞吐率与数据量大小的关系图。

上图中有一些variance的存在，并可以明显看到，吞吐率并不受磁盘上所存数据量大小的影响。实际上从上图可以看到，当磁盘数据量达到1TB时，吞吐率和磁盘数据只有几百MB时没有明显区别。

这个variance是由Linux I/O管理造成的，它会把数据缓存起来再批量flush。上图的测试结果是在生产环境中对Kafka集群做了些tuning后得到的，这些tuning方法可参考 [这里](http://kafka.apache.org/documentation.html#hwandos) 。

**consumer**吞吐率

需要注意的是，replication factor并不会影响consumer的吞吐率测试，因为consumer只会从每个partition的leader读数据，而与 replicaiton factor无关。同样，consumer吞吐率也与同步复制还是异步复制无关。

**1**个**consumer**

该测试从有6个partition，3个replication的topic消费50 million的消息。测试结果为 ***940,521 records/second***（ ***89.7MB/second***）。

可以看到，Kafkar的consumer是非常高效的。它直接从broker的文件系统里读取文件块。Kafka使用 [sendfile API](http://www.ibm.com/developerworks/library/j-zerocopy/) 来直接通过操作系统直接传输，而不用把数据拷贝到用户空间。该项测试实际上从log的起始处开始读数据，所以它做了真实的I/O。在生产环境下，consumer可以直接读取producer刚刚写下的数据（它可能还在缓存中）。实际上，如果在生产环境下跑 [I/O stat](http://en.wikipedia.org/wiki/Iostat) ，你可以看到基本上没有物理“读”。也就是说生产环境下consumer的吞吐率会比该项测试中的要高。

**3**个**consumer**

将上面的consumer复制到3台不同的机器上，并且并行运行它们（从同一个topic上消费数据）。测试结果为 ***2,615,968 records/second***（ ***249.5MB/second***）。

正如所预期的那样，consumer的吞吐率几乎线性增涨。

**Producer and Consumer**

上面的测试只是把producer和consumer分开测试，而该项测试同时运行producer和consumer，这更接近使用场景。实际上目前的replication系统中follower就相当于consumer在工作。

该项测试，在具有6个partition和3个replica的topic上同时使用1个producer和1个consumer，并且使用异步复制。测试结果为 ***795,064 records/second***（ ***75.8MB/second***）。

可以看到，该项测试结果与单独测试1个producer时的结果几乎一致。所以说consumer非常轻量级。

消息长度对吞吐率的影响

上面的所有测试都基于短消息（payload 100字节），而正如上文所说，短消息对Kafka来说是更难处理的使用方式，可以预期，随着消息长度的增大，records/second会减小，但 MB/second会有所提高。下图是records/second与消息长度的关系图。

正如我们所预期的那样，随着消息长度的增加，每秒钟所能发送的消息的数量逐渐减小。但是如果看每秒钟发送的消息的总大小，它会随着消息长度的增加而增加，如下图所示。

从上图可以看出，当消息长度为10字节时，因为要频繁入队，花了太多时间获取锁，CPU成了瓶颈，并不能充分利用带宽。但从100字节开始，我们可以看到带宽的使用逐渐趋于饱和（虽然MB/second还是会随着消息长度的增加而增加，但增加的幅度也越来越小）。

端到端的**Latency**

上文中讨论了吞吐率，那消息传输的latency如何呢？也就是说消息从producer到consumer需要多少时间呢？该项测试创建1个producer和1个consumer并反复计时。结果是， *2 ms (median), 3ms (99th percentile, 14ms (99.9th percentile)* 。

（这里并没有说明topic有多少个partition，也没有说明有多少个replica，replication是同步还是异步。实际上这 会极大影响producer发送的消息被commit的latency，而只有committed的消息才能被consumer所消费，所以它会最终影响 端到端的latency）

重现该**benchmark**

如果读者想要在自己的机器上重现本次benchmark测试，可以参考 [本次测试的配置和所使用的命令](https://gist.github.com/jkreps/c7ddb4041ef62a900e6c) 。

实际上Kafka Distribution提供了producer性能测试工具，可通过 bin/kafka-producer-perf-test.sh 脚本来启动。所使用的命令如下

Producer

Setup

bin/kafka-topics.sh --zookeeper esv4-hcl197.grid.linkedin.com:2181 --create --topic test-rep-one --partitions 6 --replication-factor 1

bin/kafka-topics.sh --zookeeper esv4-hcl197.grid.linkedin.com:2181 --create --topic test --partitions 6 --replication-factor 3

Single thread, no replication

bin/kafka-run-class.sh org.apache.kafka.clients.tools.ProducerPerformance test7 50000000 100 -1 acks=1 bootstrap.servers=esv4-hcl198.grid.linkedin.com:9092 buffer.memory=67108864 batch.size=8196

Single-thread, async 3x replication

bin/kafktopics.sh --zookeeper esv4-hcl197.grid.linkedin.com:2181 --create --topic test --partitions 6 --replication-factor 3

bin/kafka-run-class.sh org.apache.kafka.clients.tools.ProducerPerformance test6 50000000 100 -1 acks=1 bootstrap.servers=esv4-hcl198.grid.linkedin.com:9092 buffer.memory=67108864 batch.size=8196

Single-thread, sync 3x replication

bin/kafka-run-class.sh org.apache.kafka.clients.tools.ProducerPerformance test 50000000 100 -1 acks=-1 bootstrap.servers=esv4-hcl198.grid.linkedin.com:9092 buffer.memory=67108864 batch.size=64000

Three Producers, 3x async replication

bin/kafka-run-class.sh org.apache.kafka.clients.tools.ProducerPerformance test 50000000 100 -1 acks=1 bootstrap.servers=esv4-hcl198.grid.linkedin.com:9092 buffer.memory=67108864 batch.size=8196

Throughput Versus Stored Data

bin/kafka-run-class.sh org.apache.kafka.clients.tools.ProducerPerformance test 50000000000 100 -1 acks=1 bootstrap.servers=esv4-hcl198.grid.linkedin.com:9092 buffer.memory=67108864 batch.size=8196

Effect of message size

for i in 10 100 1000 10000 100000;

do

echo ""

echo $i

bin/kafka-run-class.sh org.apache.kafka.clients.tools.ProducerPerformance test $((1000\*1024\*1024/$i)) $i -1 acks=1 bootstrap.servers=esv4-hcl198.grid.linkedin.com:9092 buffer.memory=67108864 batch.size=128000

done;

Consumer

Consumer throughput

bin/kafka-consumer-perf-test.sh --zookeeper esv4-hcl197.grid.linkedin.com:2181 --messages 50000000 --topic test --threads 1

3 Consumers

On three servers, run:

bin/kafka-consumer-perf-test.sh --zookeeper esv4-hcl197.grid.linkedin.com:2181 --messages 50000000 --topic test --threads 1

End-to-end Latency

bin/kafka-run-class.sh kafka.tools.TestEndToEndLatency esv4-hcl198.grid.linkedin.com:9092 esv4-hcl197.grid.linkedin.com:2181 test 5000

Producer and consumer

bin/kafka-run-class.sh org.apache.kafka.clients.tools.ProducerPerformance test 50000000 100 -1 acks=1 bootstrap.servers=esv4-hcl198.grid.linkedin.com:9092 buffer.memory=67108864 batch.size=8196

bin/kafka-consumer-perf-test.sh --zookeeper esv4-hcl197.grid.linkedin.com:2181 --messages 50000000 --topic test --threads 1

broker配置如下

############################# Server Basics #############################

# The id of the broker. This must be set to a unique integer for each broker.

broker.id=0

############################# Socket Server Settings #############################

# The port the socket server listens on

port=9092

# Hostname the broker will bind to and advertise to producers and consumers.

# If not set, the server will bind to all interfaces and advertise the value returned from

# from java.net.InetAddress.getCanonicalHostName().

#host.name=localhost

# The number of threads handling network requests

num.network.threads=4

# The number of threads doing disk I/O

num.io.threads=8

# The send buffer (SO\_SNDBUF) used by the socket server

socket.send.buffer.bytes=1048576

# The receive buffer (SO\_RCVBUF) used by the socket server

socket.receive.buffer.bytes=1048576

# The maximum size of a request that the socket server will accept (protection against OOM)

socket.request.max.bytes=104857600

############################# Log Basics #############################

# The directory under which to store log files

log.dirs=/grid/a/dfs-data/kafka-logs,/grid/b/dfs-data/kafka-logs,/grid/c/dfs-data/kafka-logs,/grid/d/dfs-data/kafka-logs,/grid/e/dfs-data/kafka-logs,/grid/f/dfs-data/kafka-logs

# The number of logical partitions per topic per server. More partitions allow greater parallelism

# for consumption, but also mean more files.

num.partitions=8

############################# Log Flush Policy #############################

# The following configurations control the flush of data to disk. This is the most

# important performance knob in kafka.

# There are a few important trade-offs here:

# 1. Durability: Unflushed data is at greater risk of loss in the event of a crash.

# 2. Latency: Data is not made available to consumers until it is flushed (which adds latency).

# 3. Throughput: The flush is generally the most expensive operation.

# The settings below allow one to configure the flush policy to flush data after a period of time or

# every N messages (or both). This can be done globally and overridden on a per-topic basis.

# Per-topic overrides for log.flush.interval.ms

#log.flush.intervals.ms.per.topic=topic1:1000, topic2:3000

############################# Log Retention Policy #############################

# The following configurations control the disposal of log segments. The policy can

# be set to delete segments after a period of time, or after a given size has accumulated.

# A segment will be deleted whenever \*either\* of these criteria are met. Deletion always happens

# from the end of the log.

# The minimum age of a log file to be eligible for deletion

log.retention.hours=168

# A size-based retention policy for logs. Segments are pruned from the log as long as the remaining

# segments don't drop below log.retention.bytes.

#log.retention.bytes=1073741824

# The maximum size of a log segment file. When this size is reached a new log segment will be created.

log.segment.bytes=536870912

# The interval at which log segments are checked to see if they can be deleted according

# to the retention policies

log.cleanup.interval.mins=1

############################# Zookeeper #############################

# Zookeeper connection string (see zookeeper docs for details).

# This is a comma separated host:port pairs, each corresponding to a zk

# server. e.g. "127.0.0.1:3000,127.0.0.1:3001,127.0.0.1:3002".

# You can also append an optional chroot string to the urls to specify the

# root directory for all kafka znodes.

zookeeper.connect=esv4-hcl197.grid.linkedin.com:2181

# Timeout in ms for connecting to zookeeper

zookeeper.connection.timeout.ms=1000000

# metrics reporter properties

kafka.metrics.polling.interval.secs=5

kafka.metrics.reporters=kafka.metrics.KafkaCSVMetricsReporter

kafka.csv.metrics.dir=/tmp/kafka\_metrics

# Disable csv reporting by default.

kafka.csv.metrics.reporter.enabled=false

replica.lag.max.messages=10000000