Producers  
  
Producer将消息发布到它指定的topic中,并负责决定发布到哪个分区。通常简单的由负载均衡机制随机选择分区，但也可以通过特定的分区函数选择分区。使用的更多的是第二种。  
  
相比传统的消息系统，Kafka可以很好的保证有序性。  
传统的队列在服务器上保存有序的消息，如果多个consumers同时从这个服务器消费消息，服务器就会以消息存储的顺序向consumer分发消息。虽然服务器按顺序发布消息，但是消息是被异步的分发到各consumer上，所以当消息到达时可能已经失去了原来的顺序，这意味着并发消费将导致顺序错乱。为了避免故障，这样的消息系统通常使用“专用consumer”的概念，其实就是只允许一个消费者消费消息，当然这就意味着失去了并发性。  
  
在这方面Kafka做的更好，通过分区的概念，Kafka可以在多个consumer组并发的情况下提供较好的有序性和负载均衡。将每个分区分只分发给一个consumer组，这样一个分区就只被这个组的一个consumer消费，就可以顺序的消费这个分区的消息。因为有多个分区，依然可以在多个consumer组之间进行负载均衡。注意consumer组的数量不能多于分区的数量，也就是有多少分区就允许多少并发消费。  
  
Kafka只能保证一个分区之内消息的有序性，在不同的分区之间是不可以的，这已经可以满足大部分应用的需求。如果需要topic中所有消息的有序性，那就只能让这个topic只有一个分区，当然也就只有一个consumer组消费它。

高级别的consumer  
  
下面是比较负载的发送接收的程序：

1. package com.sohu.kafkademon;
2. import java.util.HashMap;
3. import java.util.List;
4. import java.util.Map;
5. import java.util.Properties;
6. import kafka.consumer.ConsumerConfig;
7. import kafka.consumer.ConsumerIterator;
8. import kafka.consumer.KafkaStream;
9. import kafka.javaapi.consumer.ConsumerConnector;
10. /\*\*
11. \* @author leicui bourne\_cui@163.com
12. \*/
13. public class KafkaConsumer extends Thread
14. {
15. private final ConsumerConnector consumer;
16. private final String topic;
17. public KafkaConsumer(String topic)
18. {
19. consumer = kafka.consumer.Consumer.createJavaConsumerConnector(
20. createConsumerConfig());
21. this.topic = topic;
22. }
23. private static ConsumerConfig createConsumerConfig()
24. {
25. Properties props = new Properties();
26. props.put("zookeeper.connect", KafkaProperties.zkConnect);
27. props.put("group.id", KafkaProperties.groupId);
28. props.put("zookeeper.session.timeout.ms", "40000");
29. props.put("zookeeper.sync.time.ms", "200");
30. props.put("auto.commit.interval.ms", "1000");
31. return new ConsumerConfig(props);
32. }
33. @Override
34. public void run() {
35. Map<String, Integer> topicCountMap = new HashMap<String, Integer>();
36. topicCountMap.put(topic, new Integer(1));
37. Map<String, List<KafkaStream<byte[], byte[]>>> consumerMap = consumer.createMessageStreams(topicCountMap);
38. KafkaStream<byte[], byte[]> stream = consumerMap.get(topic).get(0);
39. ConsumerIterator<byte[], byte[]> it = stream.iterator();
40. while (it.hasNext()) {
41. System.out.println("receive：" + new String(it.next().message()));
42. try {
43. sleep(3000);
44. } catch (InterruptedException e) {
45. e.printStackTrace();
46. }
47. }
48. }
49. }

另外再来讨论一下JVM,以下两个事实是众所周知的：  
  
•Java对象占用空间是非常大的，差不多是要存储的数据的两倍甚至更高。  
  
•随着堆中数据量的增加，垃圾回收回变的越来越困难。  
  
基于以上分析，如果把数据缓存在内存里，因为需要存储两份，不得不使用两倍的内存空间，Kafka基于JVM，又不得不将空间再次加倍,再加上要避免GC带来的性能影响，在一个32G内存的机器上，不得不使用到28-30G的内存空间。并且当系统重启的时候，又必须要将数据刷到内存中（ 10GB 内存差不多要用10分钟），就算使用冷刷新（不是一次性刷进内存，而是在使用数据的时候没有就刷到内存）也会导致最初的时候新能非常慢。但是使用文件系统，即使系统重启了，也不需要刷新数据。使用文件系统也简化了维护数据一致性的逻辑。  
  
所以与传统的将数据缓存在内存中然后刷到硬盘的设计不同，Kafka直接将数据写到了文件系统的日志中。  
  
常量时间的操作效率  
  
在大多数的消息系统中，数据持久化的机制往往是为每个cosumer提供一个B树或者其他的随机读写的数据结构。B树当然是很棒的，但是也带了一些代价：比如B树的复杂度是O(log N)，O(log N)通常被认为就是常量复杂度了，但对于硬盘操作来说并非如此。磁盘进行一次搜索需要10ms，每个硬盘在同一时间只能进行一次搜索，这样并发处理就成了问题。虽然存储系统使用缓存进行了大量优化，但是对于树结构的性能的观察结果却表明，它的性能往往随着数据的增长而线性下降，数据增长一倍，速度就会降低一倍。  
  
直观的讲，对于主要用于日志处理的消息系统，数据的持久化可以简单的通过将数据追加到文件中实现，读的时候从文件中读就好了。这样做的好处是读和写都是 O(1) 的，并且读操作不会阻塞写操作和其他操作。这样带来的性能优势是很明显的，因为性能和数据的大小没有关系了。  
  
既然可以使用几乎没有容量限制（相对于内存来说）的硬盘空间建立消息系统，就可以在没有性能损失的情况下提供一些一般消息系统不具备的特性。比如，一般的消息系统都是在消息被消费后立即删除，Kafka却可以将消息保存一段时间（比如一星期），这给consumer提供了很好的机动性和灵活性，这点在今后的文章中会有详述。  
  
############################################################  
  
五、消息传输的事务定义  
  
之前讨论了consumer和producer是怎么工作的，现在来讨论一下数据传输方面。数据传输的事务定义通常有以下三种级别：

1. 最多一次: 消息不会被重复发送，最多被传输一次，但也有可能一次不传输。
2. 最少一次: 消息不会被漏发送，最少被传输一次，但也有可能被重复传输.
3. 精确的一次（Exactly once）: 不会漏传输也不会重复传输,每个消息都传输被一次而且仅仅被传输一次，这是大家所期望的。

大多数消息系统声称可以做到“精确的一次”，但是仔细阅读它们的的文档可以看到里面存在误导，比如没有说明当consumer或producer失败时怎么样，或者当有多个consumer并行时怎么样，或写入硬盘的数据丢失时又会怎么样。kafka的做法要更先进一些。当发布消息时，Kafka有一个“committed”的概念，一旦消息被提交了，只要消息被写入的分区的所在的副本broker是活动的，数据就不会丢失。关于副本的活动的概念，下节文档会讨论。现在假设broker是不会down的。  
  
如果producer发布消息时发生了网络错误，但又不确定实在提交之前发生的还是提交之后发生的，这种情况虽然不常见，但是必须考虑进去，现在Kafka版本还没有解决这个问题，将来的版本正在努力尝试解决。  
  
并不是所有的情况都需要“精确的一次”这样高的级别，Kafka允许producer灵活的指定级别。比如producer可以指定必须等待消息被提交的通知，或者完全的异步发送消息而不等待任何通知，或者仅仅等待leader声明它拿到了消息（followers没有必要）。  
  
现在从consumer的方面考虑这个问题，所有的副本都有相同的日志文件和相同的offset，consumer维护自己消费的消息的offset，如果consumer不会崩溃当然可以在内存中保存这个值，当然谁也不能保证这点。如果consumer崩溃了，会有另外一个consumer接着消费消息，它需要从一个合适的offset继续处理。这种情况下可以有以下选择：

* consumer可以先读取消息，然后将offset写入日志文件中，然后再处理消息。这存在一种可能就是在存储offset后还没处理消息就crash了，新的consumer继续从这个offset处理，那么就会有些消息永远不会被处理，这就是上面说的“最多一次”。
* consumer可以先读取消息，处理消息，最后记录offset，当然如果在记录offset之前就crash了，新的consumer会重复的消费一些消息，这就是上面说的“最少一次”。
* “精确一次”可以通过将提交分为两个阶段来解决：保存了offset后提交一次，消息处理成功之后再提交一次。但是还有个更简单的做法：将消息的offset和消息被处理后的结果保存在一起。比如用Hadoop ETL处理消息时，将处理后的结果和offset同时保存在HDFS中，这样就能保证消息和offser同时被处理了。

############################################################

Kafka Producer消息发送  
producer直接将数据发送到broker的leader(主节点)，不需要在多个节点进行分发。为了帮助producer做到这点，所有的Kafka节点都可以及时的告知:哪些节点是活动的，目标topic目标分区的leader在哪。这样producer就可以直接将消息发送到目的地了。  
  
客户端控制消息将被分发到哪个分区。可以通过负载均衡随机的选择，或者使用分区函数。Kafka允许用户实现分区函数，指定分区的key，将消息hash到不同的分区上(当然有需要的话，也可以覆盖这个分区函数自己实现逻辑).比如如果你指定的key是user id，那么同一个用户发送的消息都被发送到同一个分区上。经过分区之后，consumer就可以有目的的消费某个分区的消息。  
  
异步发送  
批量发送可以很有效的提高发送效率。Kafka producer的异步发送模式允许进行批量发送，先将消息缓存在内存中，然后一次请求批量发送出去。这个策略可以配置的，比如可以指定缓存的消息达到某个量的时候就发出去，或者缓存了固定的时间后就发送出去（比如100条消息就发送，或者每5秒发送一次）。这种策略将大大减少服务端的I/O次数。  
  
既然缓存是在producer端进行的，那么当producer崩溃时，这些消息就会丢失。Kafka0.8.1的异步发送模式还不支持回调，就不能在发送出错时进行处理。Kafka 0.9可能会增加这样的回调函数。见[Proposed Producer API](https://cwiki.apache.org/confluence/display/KAFKA/Client+Rewrite#ClientRewrite-ProposedProducerAPI).  
  
Kafka Consumer  
Kafa consumer消费消息时，向broker发出"fetch"请求去消费特定分区的消息。consumer指定消息在日志中的偏移量（offset），就可以消费从这个位置开始的消息。customer拥有了offset的控制权，可以向后回滚去重新消费之前的消息，这是很有意义的。  
  
推还是拉？  
Kafka最初考虑的问题是，customer应该从brokes拉取消息还是brokers将消息推送到consumer，也就是pull还push。在这方面，Kafka遵循了一种大部分消息系统共同的传统的设计：producer将消息推送到broker，consumer从broker拉取消息。  
  
一些消息系统比如Scribe和Apache Flume采用了push模式，将消息推送到下游的consumer。这样做有好处也有坏处：由broker决定消息推送的速率，对于不同消费速率的consumer就不太好处理了。消息系统都致力于让consumer以最大的速率最快速的消费消息，但不幸的是，push模式下，当broker推送的速率远大于consumer消费的速率时，consumer恐怕就要崩溃了。最终Kafka还是选取了传统的pull模式。  
  
Pull模式的另外一个好处是consumer可以自主决定是否批量的从broker拉取数据。Push模式必须在不知道下游consumer消费能力和消费策略的情况下决定是立即推送每条消息还是缓存之后批量推送。如果为了避免consumer崩溃而采用较低的推送速率，将可能导致一次只推送较少的消息而造成浪费。Pull模式下，consumer就可以根据自己的消费能力去决定这些策略。  
  
Pull有个缺点是，如果broker没有可供消费的消息，将导致consumer不断在循环中轮询，直到新消息到t达。为了避免这点，Kafka有个参数可以让consumer阻塞知道新消息到达(当然也可以阻塞知道消息的数量达到某个特定的量这样就可以批量发送)。  
  
消费状态跟踪  
对消费消息状态的记录也是很重要的。  
大部分消息系统在broker端的维护消息被消费的记录：一个消息被分发到consumer后broker就马上进行标记或者等待customer的通知后进行标记。这样也可以在消息在消费后立马就删除以减少空间占用。  
  
但是这样会不会有什么问题呢？如果一条消息发送出去之后就立即被标记为消费过的，一旦consumer处理消息时失败了（比如程序崩溃）消息就丢失了。为了解决这个问题，很多消息系统提供了另外一个个功能：当消息被发送出去之后仅仅被标记为已发送状态，当接到consumer已经消费成功的通知后才标记为已被消费的状态。这虽然解决了消息丢失的问题，但产生了新问题，首先如果consumer处理消息成功了但是向broker发送响应时失败了，这条消息将被消费两次。第二个问题时，broker必须维护每条消息的状态，并且每次都要先锁住消息然后更改状态然后释放锁。这样麻烦又来了，且不说要维护大量的状态数据，比如如果消息发送出去但没有收到消费成功的通知，这条消息将一直处于被锁定的状态，  
Kafka采用了不同的策略。Topic被分成了若干分区，每个分区在同一时间只被一个consumer消费。这意味着每个分区被消费的消息在日志中的位置仅仅是一个简单的整数：offset。这样就很容易标记每个分区消费状态就很容易了，仅仅需要一个整数而已。这样消费状态的跟踪就很简单了。  
  
这带来了另外一个好处：consumer可以把offset调成一个较老的值，去重新消费老的消息。这对传统的消息系统来说看起来有些不可思议，但确实是非常有用的，谁规定了一条消息只能被消费一次呢？consumer发现解析数据的程序有bug，在修改bug后再来解析一次消息，看起来是很合理的额呀！  
  
离线处理消息  
高级的数据持久化允许consumer每个隔一段时间批量的将数据加载到线下系统中比如[Hadoop](http://www.linuxidc.com/topicnews.aspx?tid=13)或者数据仓库。这种情况下，Hadoop可以将加载任务分拆，拆成每个broker或每个topic或每个分区一个加载任务。Hadoop具有任务管理功能，当一个任务失败了就可以重启而不用担心数据被重新加载，只要从上次加载的位置继续加载消息就可以了。  
  
#########################################################  
  
  
八、主从同步  
  
  
Kafka允许topic的分区拥有若干副本，这个数量是可以配置的，你可以为每个topci配置副本的数量。Kafka会自动在每个个副本上备份数据，所以当一个节点down掉时数据依然是可用的。  
  
Kafka的副本功能不是必须的，你可以配置只有一个副本，这样其实就相当于只有一份数据。  
创建副本的单位是topic的分区，每个分区都有一个leader和零或多个followers.所有的读写操作都由leader处理，一般分区的数量都比broker的数量多的多，各分区的leader均匀的分布在brokers中。所有的followers都复制leader的日志，日志中的消息和顺序都和leader中的一致。flowers向普通的consumer那样从leader那里拉取消息并保存在自己的日志文件中。  
  
许多分布式的消息系统自动的处理失败的请求，它们对一个节点是否  
  
着（alive）”有着清晰的定义。Kafka判断一个节点是否活着有两个条件：

1. 节点必须可以维护和ZooKeeper的连接，Zookeeper通过心跳机制检查每个节点的连接。
2. 如果节点是个follower,他必须能及时的同步leader的写操作，延时不能太久。

符合以上条件的节点准确的说应该是“同步中的（in sync）”，而不是模糊的说是“活着的”或是“失败的”。Leader会追踪所有“同步中”的节点，一旦一个down掉了，或是卡住了，或是延时太久，leader就会把它移除。至于延时多久算是“太久”，是由参数replica.lag.max.messages决定的，怎样算是卡住了，怎是由参数replica.lag.time.max.ms决定的。   
  
只有当消息被所有的副本加入到日志中时，才算是“committed”，只有committed的消息才会发送给consumer，这样就不用担心一旦leader down掉了消息会丢失。Producer也可以选择是否等待消息被提交的通知，这个是由参数request.required.acks决定的。  
Kafka保证只要有一个“同步中”的节点，“committed”的消息就不会丢失。  
  
Leader的选择  
Kafka的核心是日志文件，日志文件在集群中的同步是分布式数据系统最基础的要素。  
  
如果leaders永远不会down的话我们就不需要followers了！一旦leader down掉了，需要在followers中选择一个新的leader.但是followers本身有可能延时太久或者crash，所以必须选择高质量的follower作为leader.必须保证，一旦一个消息被提交了，但是leader down掉了，新选出的leader必须可以提供这条消息。大部分的分布式系统采用了多数投票法则选择新的leader,对于多数投票法则，就是根据所有副本节点的状况动态的选择最适合的作为leader.Kafka并不是使用这种方法。  
  
Kafaka动态维护了一个同步状态的副本的集合（a set of in-sync replicas），简称ISR，在这个集合中的节点都是和leader保持高度一致的，任何一条消息必须被这个集合中的每个节点读取并追加到日志中了，才回通知外部这个消息已经被提交了。因此这个集合中的任何一个节点随时都可以被选为leader.ISR在ZooKeeper中维护。ISR中有f+1个节点，就可以允许在f个节点down掉的情况下不会丢失消息并正常提供服。ISR的成员是动态的，如果一个节点被淘汰了，当它重新达到“同步中”的状态时，他可以重新加入ISR.这种leader的选择方式是非常快速的，适合kafka的应用场景。  
  
一个邪恶的想法：如果所有节点都down掉了怎么办？Kafka对于数据不会丢失的保证，是基于至少一个节点是存活的，一旦所有节点都down了，这个就不能保证了。  
实际应用中，当所有的副本都down掉时，必须及时作出反应。可以有以下两种选择:

1. 等待ISR中的任何一个节点恢复并担任leader。
2. 选择所有节点中（不只是ISR）第一个恢复的节点作为leader.

这是一个在可用性和连续性之间的权衡。如果等待ISR中的节点恢复，一旦ISR中的节点起不起来或者数据都是了，那集群就永远恢复不了了。如果等待ISR意外的节点恢复，这个节点的数据就会被作为线上数据，有可能和真实的数据有所出入，因为有些数据它可能还没同步到。Kafka目前选择了第二种策略，在未来的版本中将使这个策略的选择可配置，可以根据场景灵活的选择。  
这种窘境不只Kafka会遇到，几乎所有的分布式数据系统都会遇到。  
  
副本管理  
以上仅仅以一个topic一个分区为例子进行了讨论，但实际上一个Kafka将会管理成千上万的topic分区.Kafka尽量的使所有分区均匀的分布到集群所有的节点上而不是集中在某些节点上，另外主从关系也尽量均衡这样每个几点都会担任一定比例的分区的leader.  
优化leader的选择过程也是很重要的，它决定了系统发生故障时的空窗期有多久。Kafka选择一个节点作为“controller”,当发现有节点down掉的时候它负责在游泳分区的所有节点中选择新的leader,这使得Kafka可以批量的高效的管理所有分区节点的主从关系。如果controller down掉了，活着的节点中的一个会备切换为新的controller.  
  
###################################################

Producer API提供了以下功能：

1. 可以将多个消息缓存到本地队列里，然后异步的批量发送到broker，可以通过参数producer.type=async做到。缓存的大小可以通过一些参数指定：queue.time和batch.size。一个后台线程（(kafka.producer.async.ProducerSendThread）从队列中取出数据并让kafka.producer.EventHandler将消息发送到broker，也可以通过参数event.handler定制handler，在producer端处理数据的不同的阶段注册处理器，比如可以对这一过程进行日志追踪，或进行一些监控。只需实现kafka.producer.async.CallbackHandler接口，并在callback.handler中配置。
2. 自己编写Encoder来序列化消息，只需实现下面这个接口。默认的Encoder是kafka.serializer.DefaultEncoder。
   1. interface Encoder<T> {
   2. public Message toMessage(T data);
   3. }
3. 提供了基于Zookeeper的broker自动感知能力，可以通过参数zk.connect实现。如果不使用Zookeeper，也可以使用broker.list参数指定一个静态的brokers列表，这样消息将被随机的发送到一个broker上，一旦选中的broker失败了，消息发送也就失败了。
4. 通过分区函数kafka.producer.Partitioner类对消息分区。
   1. interface Partitioner<T> {
   2. int partition(T key, int numPartitions);
   3. }

分区函数有两个参数：key和可用的分区数量，从分区列表中选择一个分区并返回id。默认的分区策略是hash(key)%numPartitions.如果key是null,就随机的选择一个。可以通过参数partitioner.class定制分区函数。

KafKa Consumer APIs  
  
Consumer API有两个级别。低级别的和一个指定的broker保持连接，并在接收完消息后关闭连接，这个级别是无状态的，每次读取消息都带着offset。  
高级别的API隐藏了和brokers连接的细节，在不必关心服务端架构的情况下和服务端通信。还可以自己维护消费状态，并可以通过一些条件指定订阅特定的topic,比如白名单黑名单或者正则表达式。  
  
低级别的API

1. class SimpleConsumer {
2. /\*向一个broker发送读取请求并得到消息集 \*/
3. public ByteBufferMessageSet fetch(FetchRequest request);
4. /\*向一个broker发送读取请求并得到一个相应集 \*/
5. public MultiFetchResponse multifetch(List<FetchRequest> fetches);
6. /\*\*
7. \* 得到指定时间之前的offsets
8. \* 返回值是offsets列表，以倒序排序
9. \* @param time: 时间，毫秒,
10. \* 如果指定为OffsetRequest.MODULE.MODULE.LATIEST\_TIME(), 得到最新的offset.
11. \* 如果指定为OffsetRequest.MODULE.MODULE.EARLIEST\_TIME(),得到最老的offset.
12. \*/
13. publiclong[] getOffsetsBefore(String topic, int partition, long time, int maxNumOffsets);
14. }

复制代码

低级别的API是高级别API实现的基础，也是为了一些对维持消费状态有特殊需求的场景，比如[Hadoop](http://www.linuxidc.com/topicnews.aspx?tid=13) consumer这样的离线consumer。  
  
高级别的API

1. /\* 创建连接 \*/
2. ConsumerConnector connector = Consumer.create(consumerConfig);
3. interface ConsumerConnector {
4. /\*\*
5. \* 这个方法可以得到一个流的列表，每个流都是MessageAndMetadata的迭代，通过MessageAndMetadata可以拿到消息和其他的元数据（目前之后topic）
6. \* Input: a map of <topic, #streams>
7. \* Output: a map of <topic, list of message streams>
8. \*/
9. public Map<String,List<KafkaStream>> createMessageStreams(Map<String,Int> topicCountMap);
10. /\*\*
11. \* 你也可以得到一个流的列表，它包含了符合TopicFiler的消息的迭代，
12. \* 一个TopicFilter是一个封装了白名单或黑名单的正则表达式。
13. \*/
14. public List<KafkaStream> createMessageStreamsByFilter(
15. TopicFilter topicFilter, int numStreams);
16. /\* 提交目前消费到的offset \*/
17. public commitOffsets()
18. /\* 关闭连接 \*/
19. public shutdown()
20. }

复制代码

这个API围绕着由KafkaStream实现的迭代器展开，每个流代表一系列从一个或多个分区多和broker上汇聚来的消息，每个流由一个线程处理，所以客户端可以在创建的时候通过参数指定想要几个流。一个流是多个分区多个broker的合并，但是每个分区的消息只会流向一个流。  
  
每调用一次createMessageStreams都会将consumer注册到topic上，这样consumer和brokers之间的负载均衡就会进行调整。API鼓励每次调用创建更多的topic流以减少这种调整。createMessageStreamsByFilter方法注册监听可以感知新的符合filter的tipic。  
  
#######################################################  
  
十、消息和日志  
  
  
  
消息由一个固定长度的头部和可变长度的字节数组组成。头部包含了一个版本号和CRC32校验码。

1. /\*\*
2. \* 具有N个字节的消息的格式如下
3. \*
4. \* 如果版本号是0
5. \*
6. \* 1. 1个字节的 "magic" 标记
7. \*
8. \* 2. 4个字节的CRC32校验码
9. \*
10. \* 3. N - 5个字节的具体信息
11. \*
12. \* 如果版本号是1
13. \*
14. \* 1. 1个字节的 "magic" 标记
15. \*
16. \* 2.1个字节的参数允许标注一些附加的信息比如是否压缩了，解码类型等
17. \*
18. \* 3.4个字节的CRC32校验码
19. \*
20. \* 4. N - 6 个字节的具体信息
21. \*
22. \*/

复制代码

日志一个叫做“my\_topic”且有两个分区的的topic,它的日志有两个文件夹组成，my\_topic\_0和my\_topic\_1,每个文件夹里放着具体的数据文件，每个数据文件都是一系列的日志实体，每个日志实体有一个4个字节的整数N标注消息的长度，后边跟着N个字节的消息。每个消息都可以由一个64位的整数offset标注，offset标注了这条消息在发送到这个分区的消息流中的起始位置。每个日志文件的名称都是这个文件第一条日志的offset.所以第一个日志文件的名字就是00000000000.kafka.所以每相邻的两个文件名字的差就是一个数字S,S差不多就是配置文件中指定的日志文件的最大容量。  
消息的格式都由一个统一的接口维护，所以消息可以在producer,broker和consumer之间无缝的传递。存储在硬盘上的消息格式如下所示：

1. 消息长度: 4 bytes (value: 1+4+n)
2. 版本号: 1 byte
3. CRC校验码: 4 bytes
4. 具体的消息: n bytes

<ignore\_js\_op>   
  
  
写操作消息被不断的追加到最后一个日志的末尾，当日志的大小达到一个指定的值时就会产生一个新的文件。对于写操作有两个参数，一个规定了消息的数量达到这个值时必须将数据刷新到硬盘上，另外一个规定了刷新到硬盘的时间间隔，这对数据的持久性是个保证，在系统崩溃的时候只会丢失一定数量的消息或者一个时间段的消息。  
  
读操作  
读操作需要两个参数：一个64位的offset和一个S字节的最大读取量。S通常比单个消息的大小要大，但在一些个别消息比较大的情况下，S会小于单个消息的大小。这种情况下读操作会不断重试，每次重试都会将读取量加倍，直到读取到一个完整的消息。可以配置单个消息的最大值，这样服务器就会拒绝大小超过这个值的消息。也可以给客户端指定一个尝试读取的最大上限，避免为了读到一个完整的消息而无限次的重试。  
在实际执行读取操纵时，首先需要定位数据所在的日志文件，然后根据offset计算出在这个日志中的offset(前面的的offset是整个分区的offset),然后在这个offset的位置进行读取。定位操作是由二分查找法完成的，Kafka在内存中为每个文件维护了offset的范围。  
  
下面是发送给consumer的结果的格式：

1. MessageSetSend (fetch result)
2. total length     : 4 bytes
3. error code       : 2 bytes
4. message 1        : x bytes
5. ...
6. message n        : x bytes
7. MultiMessageSetSend (multiFetch result)
8. total length       : 4 bytes
9. error code         : 2 bytes
10. messageSetSend 1
11. ...
12. messageSetSend n

复制代码

删除  
日志管理器允许定制删除策略。目前的策略是删除修改时间在N天之前的日志（按时间删除），也可以使用另外一个策略：保留最后的N GB数据的策略(按大小删除)。为了避免在删除时阻塞读操作，采用了copy-on-write形式的实现，删除操作进行时，读取操作的二分查找功能实际是在一个静态的快照副本上进行的，这类似于Java的CopyOnWriteArrayList。  
  
可靠性保证  
日志文件有一个可配置的参数M，缓存超过这个数量的消息将被强行刷新到硬盘。一个日志矫正线程将循环检查最新的日志文件中的消息确认每个消息都是合法的。合法的标准为：所有文件的大小的和最大的offset小于日志文件的大小，并且消息的CRC32校验码与存储在消息实体中的校验码一致。如果在某个offset发现不合法的消息，从这个offset到下一个合法的offset之间的内容将被移除。  
有两种情况必须考虑：  
1，当发生崩溃时有些数据块未能写入。  
2，写入了一些空白数据块。第二种情况的原因是，对于每个文件，操作系统都有一个inode（inode是指在许多“类Unix文件系统”中的一种数据结构。每个inode保存了文件系统中的一个文件系统对象,包括文件、目录、大小、设备文件、socket、管道, 等等），但无法保证更新inode和写入数据的顺序，当inode保存的大小信息被更新了，但写入数据时发生了崩溃，就产生了空白数据块。CRC校验码可以检查这些块并移除，当然因为崩溃而未写入的数据块也就丢失了。