转 KAFKA：如何做到1秒发布百万级条消息

出处:
http://rdcqii.hundsun.com/portal/article/709.html

KAFKA是分布式发布-订阅消息系统，是一个分布式的，可划分的，冗余备份的持久性的日志服务。它主要用于处理活跃的流式数据。

现在被广泛地应用于构建实时数据管道和流应用的场景中，具有横向扩展，容错，快等优点，并已经运行在众多大中型公司的生产环境中，成功应用于大数据领域，本文分享一下我所了解的KAFKA。

1 KAFKA高吞吐率性能揭秘

KAFKA的第一个突出特定就是“快”，而且是那种变态的“快”，在普通廉价的虚拟机器上，比如一般SAS盘做的虚拟机上，据LINDEDIN统计，最新的数据是每天利用KAFKA处理的消息超过1万亿条，在峰值时每秒钟会发布超过百万条消息，就算是在内存和CPU都不高的情况下，Kafka的速度最高可以达到每秒十万条数据，并且还能持久化存储。

作为消息队列，要承接读跟写两块的功能，首先是写，就是消息日志写入KAFKA，那么，KAFKA在“写”上是怎么做到写变态快呢?

1.1 KAFKA让代码飞起来之写得快

首先，可以使用KAFKA提供的生产端API发布消息到1个或多个Topic（主题）的一个（保证数据的顺序）或者多个分区（并行处理，但不一定保证数据顺序）。Topic可以简单理解成一个数据类别，是用来区分不同数据的。

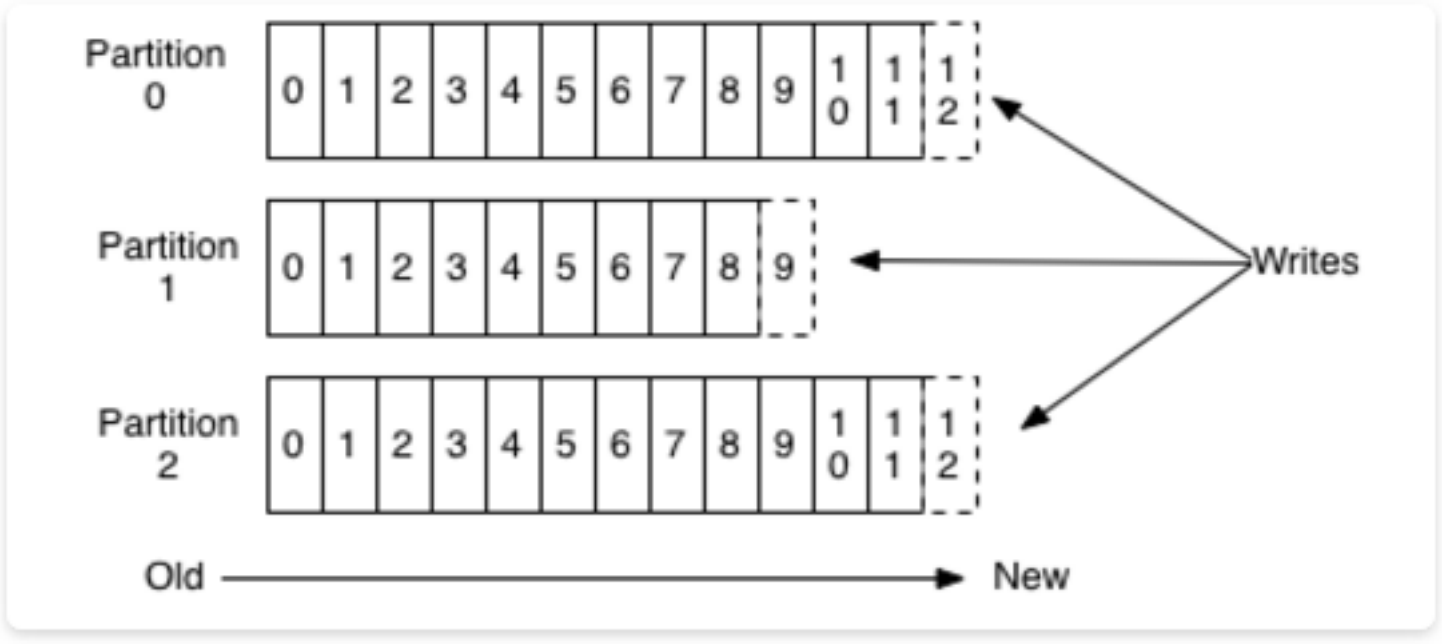
KAFKA维护一个Topic中的分区log，以顺序追加的方式向各个分区中写入消息，每个分区都是不可变的消息队列。分区中的消息都是以k-v形式存在。

- k表示offset，称之为偏移量，一个64位整型的唯一标识，offset代表了Topic分区中所有消息流中该消息的起始字节位置。
- v就是实际的消息内容，每个分区中的每个offset都是唯一存在的，所有分

区的消息都是一次写入，在消息未过期之前都可以调整offset来实现多次读取。

以上提到KAFKA“快”的第一个因素：消息顺序写入磁盘。

我们知道现在的磁盘大多数都还是机械结构（SSD不在讨论的范围内），如果将消息以随机写的方式存入磁盘，就会按柱面、磁头、扇区的方式进行（寻址过程），缓慢的机械运动（相对内存）会消耗大量时间，导致磁盘的写入速度只能达到内存写入速度的几百万分之一，为了规避随机写带来的时间消耗，KAFKA采取顺序写的方式存储数据，如下图所示：



Kafka消息写入

新来的消息只能追加到已有消息的末尾，并且已经生产的消息不支持随机删除以及随机访问，但是消费者可以通过重置offset的方式来访问已经消费过的数据。

即使顺序读写，过于频繁的大量小I/O操作一样会造成磁盘的瓶颈，所以KAFKA在此处的处理是把这些消息集合在一起批量发送，这样减少对磁盘IO的过度读写，而不是一次发送单个消息。

另一个是无效率的字节复制，尤其是在负载比较高的情况下影响是显着的。为了避免这种情况，KAFKA采用由Producer，broker和consumer共享的标准化二进制消息格式，这样数据块就可以在它们之间自由传输，无需转换，降低了字节复制的成本开销。

同时，KAFKA采用了MMAP(Memory Mapped Files，内存映射文件)技术。很

多现代操作系统都大量使用主存做磁盘缓存，一个现代操作系统可以将内存中的所有剩余空间用作磁盘缓存，而当内存回收的时候几乎没有性能损失。

由于KAFKA是基于JVM的，并且任何与Java内存使用打过交道的人都知道两件事：

- 对象的内存开销非常高，通常是实际要存储数据大小的两倍；
- 随着数据的增加，java的垃圾收集也会越来越频繁并且缓慢。

基于此，使用文件系统，同时依赖页面缓存就比使用其他数据结构和维护内存缓存更有吸引力：

- 不使用进程内缓存，就腾出了内存空间，可以用来存放页面缓存的空间几乎可以翻倍。
- 如果KAFKA重启，进行内缓存就会丢失，但是使用操作系统的页面缓存依然可以继续使用。

可能有人会问KAFKA如此频繁利用页面缓存，如果内存大小不够了怎么办?

KAFKA会将数据写入到持久化日志中而不是刷新到磁盘。实际上它只是转移到了内核的页面缓存。

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，它可以直接利用操作系统的页缓存来实现文件到物理内存的直接映射。完成映射之后对物理内存的操作在适当时候会被同步到硬盘上。

1.2 KAFKA让代码飞起来之读得快

KAFKA除了接收数据时写得快，另外一个特点就是推送数据时发得快。

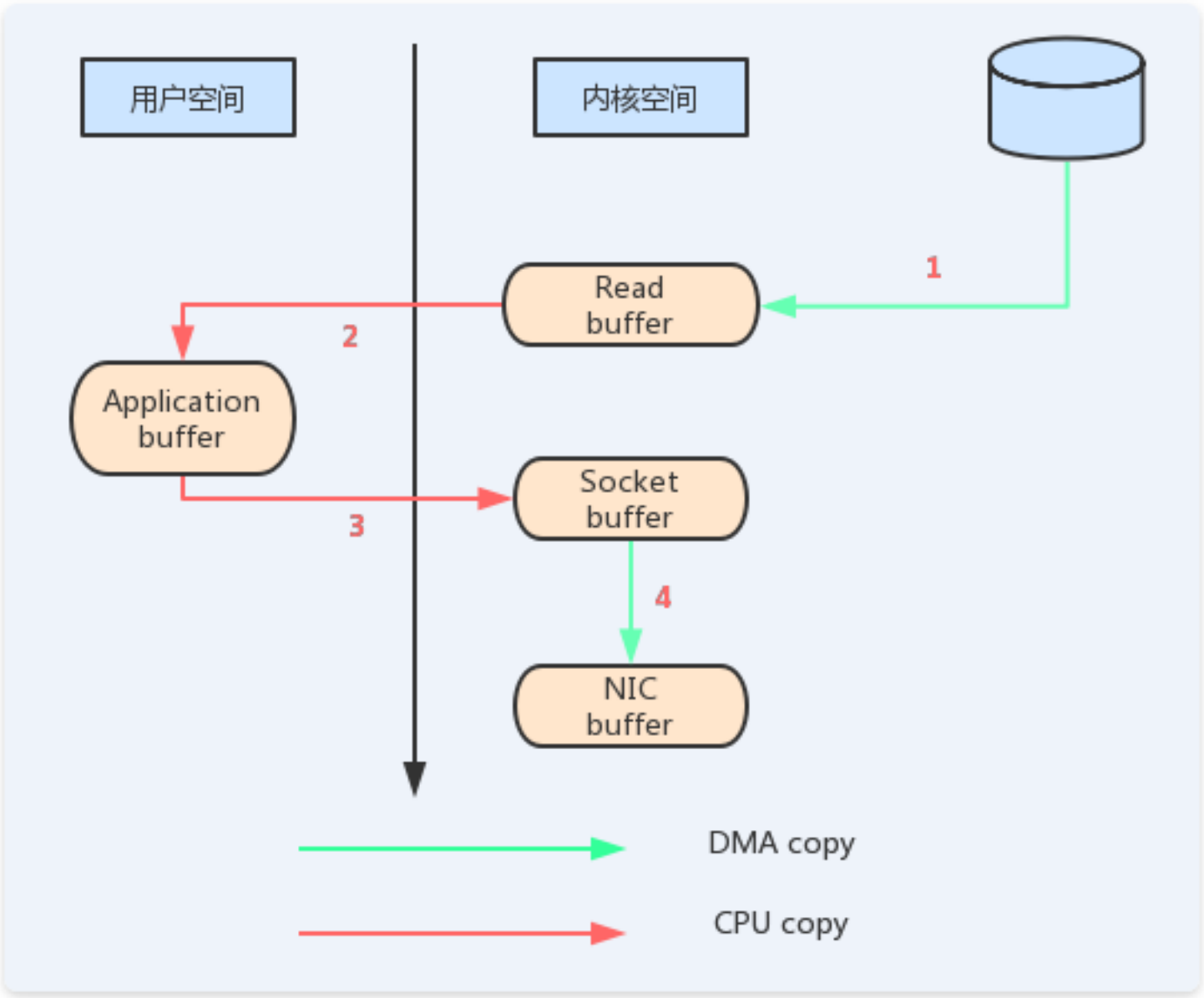
KAFKA这种消息队列在生产端和消费端分别采取的push和pull的方式，也就是你生产端可以认为KAFKA是个无底洞，有多少数据可以使劲往里面推送，消费端则是根据自己的消费能力，需要多少数据，你自己过来KAFKA这里拉取，KAFKA能保证只要这里有数据，消费端需要多少，都尽可以自己过来拿。

▲零拷贝

具体到消息的落地保存，broker维护的消息日志本身就是文件的目录，每个

文件都是二进制保存，生产者和消费者使用相同的格式来处理。维护这个公共的格式并允许优化最重要的操作：网络传输持久性日志块。 现代的unix操作系统提供一个优化的代码路径，用于将数据从页缓存传输到socket；在Linux中，是通过sendfile系统调用来完成的。Java提供了访问这个系统调用的方法：FileChannel.transferTo API。

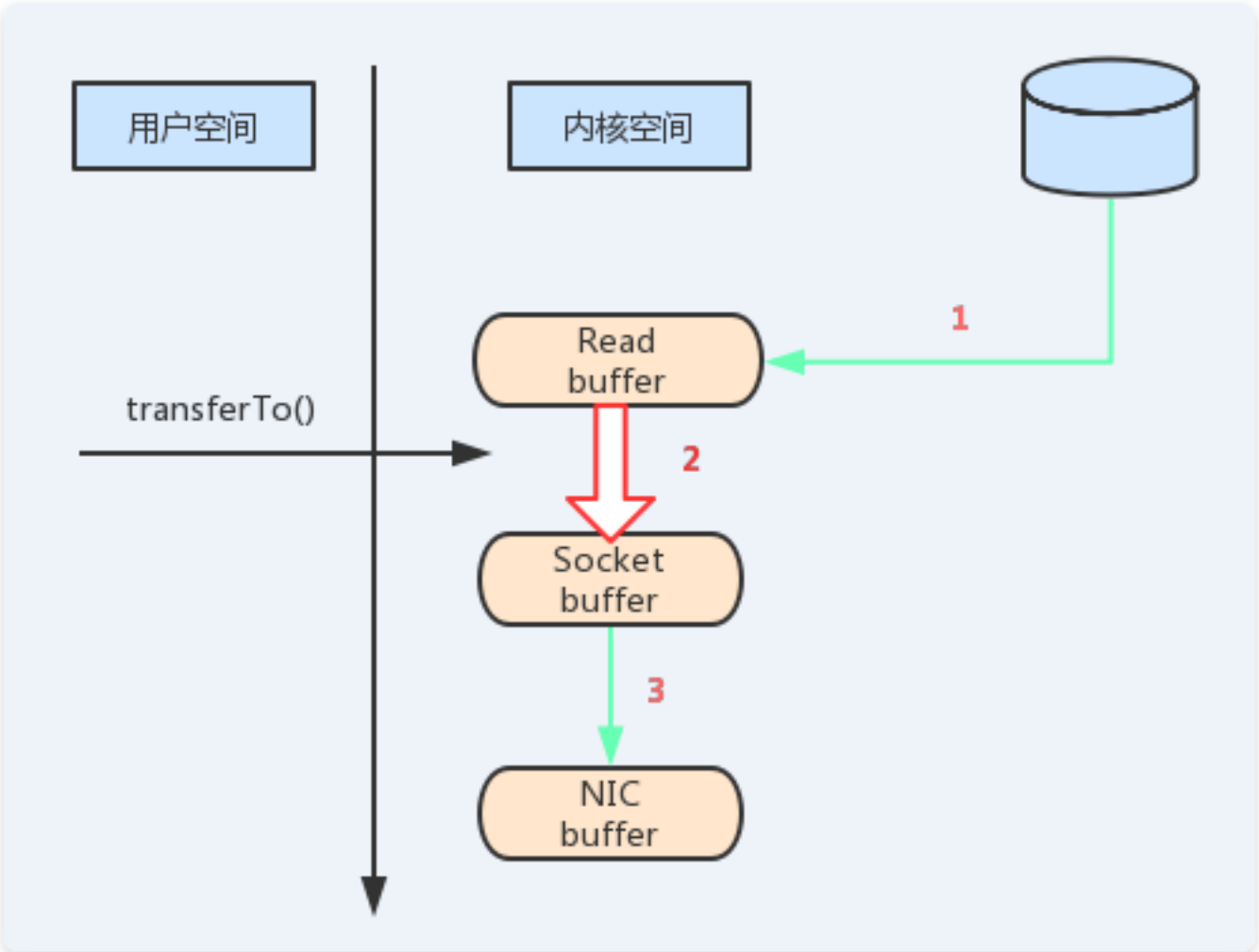
要理解senfile的影响，重要的是要了解将数据从文件传输到socket的公共数据路径，如下图所示，数据从磁盘传输到socket要经过以下几个步骤：



文件传输到socket的常规方式

- 操作系统将数据从磁盘读入到内核空间的页缓存
- 应用程序将数据从内核空间读入到用户空间缓存中
- 应用程序将数据写回到内核空间到socket缓存中
- 操作系统将数据从socket缓冲区复制到网卡缓冲区，以便将数据经网络发出

这里有四次拷贝，两次系统调用，这是非常低效的做法。如果使用sendfile，只需要一次拷贝就行：允许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。



零拷贝方式传输文件到socket

常规文件传输和zeroCopy方式的性能对比：

File size	Normal file transfer (ms)	transferTo (ms)
7MB	156	45
21MB	337	128
63MB	843	387
98MB	1320	617
200MB	2124	1150
350MB	3631	1762
700MB	13498	4422
1GB	18399	8537

常规文件传输和zeroCopy方式的性能对比

假设一个Topic有多个消费者的情况， 并使用上面的零拷贝优化，数据被复制到页缓存中一次，并在每个消费上重复使用，而不是存储在存储器中，也不在每次读取时复制到用户空间。 这使得以接近网络连接限制的速度消费消息。

这种页缓存和sendfile组合，意味着KAFKA集群的消费者大多数都完全从缓存消费消息，而磁盘没有任何读取活动。

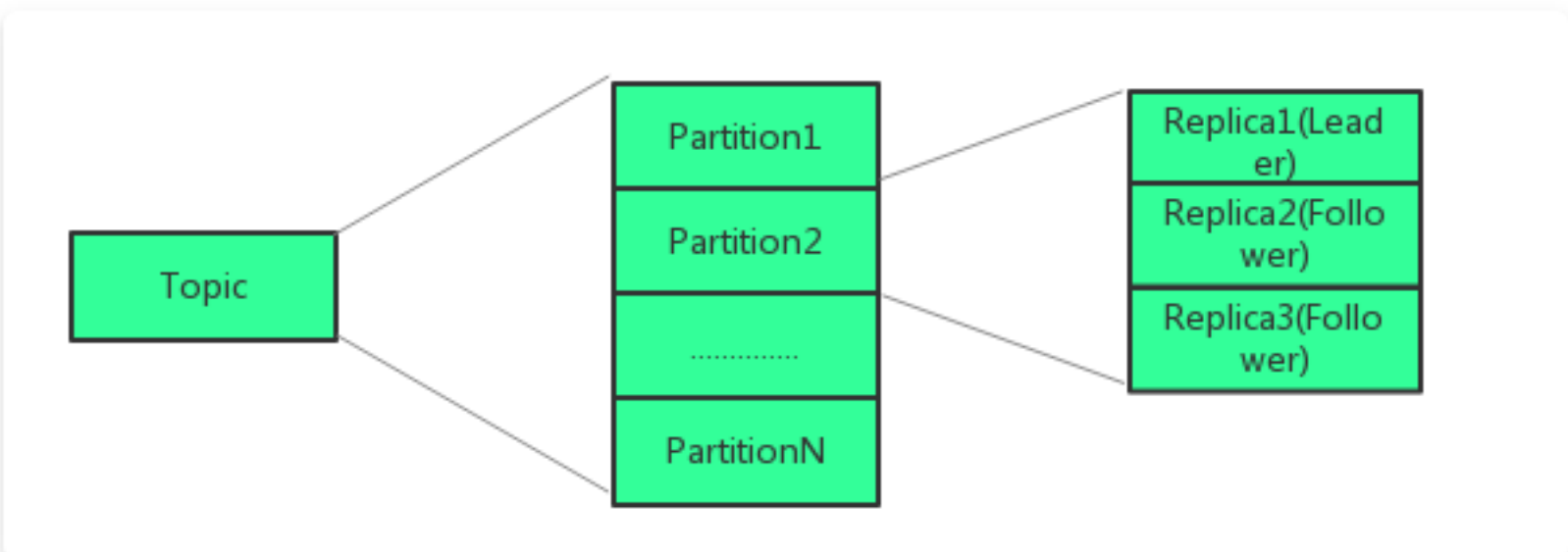
▲批量压缩

在很多情况下，系统的瓶颈不是CPU或磁盘，而是网络带宽，对于需要在广域网上的数据中心之间发送消息的数据流水线尤其如此。所以数据压缩就很重要。可以每个消息都压缩，但是压缩率相对很低。所以KAFKA使用了批量压缩，即将多个消息一起压缩而不是单个消息压缩。

KAFKA允许使用递归的消息集合，批量的消息可以通过压缩的形式传输并且在日志中也可以保持压缩格式，直到被消费者解压缩。

KAFKA支持Gzip和Snappy压缩协议。

2 KAFKA数据可靠性深度解读

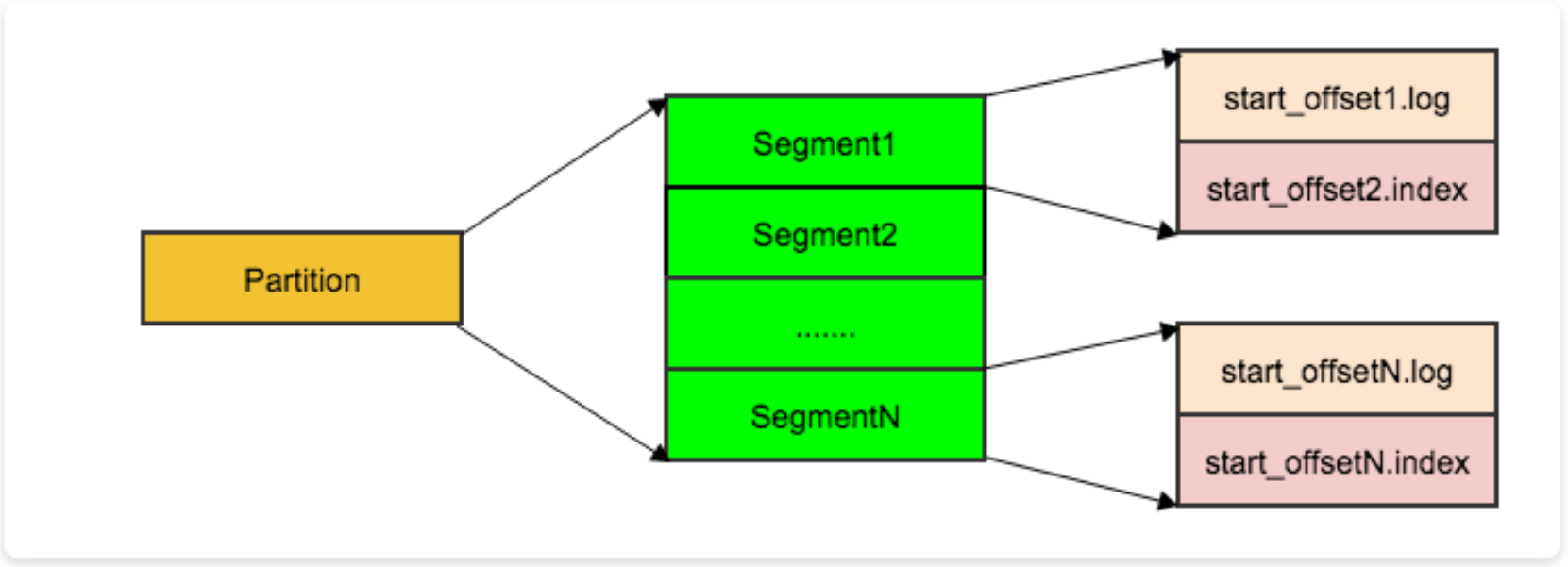


Topic逻辑结构

KAFKA的消息保存在Topic中，Topic可分为多个分区，为保证数据的安全性，每个分区又有多个Replia。

- 多分区的设计的特点：

1.为了并发读写，加快读写速度；
2.是利用多分区的存储，利于数据的均衡；
3.是为了加快数据的恢复速率，一但某台机器挂了，整个集群只需要恢复一部分数据，可加快故障恢复的时间。



Partition存储结构

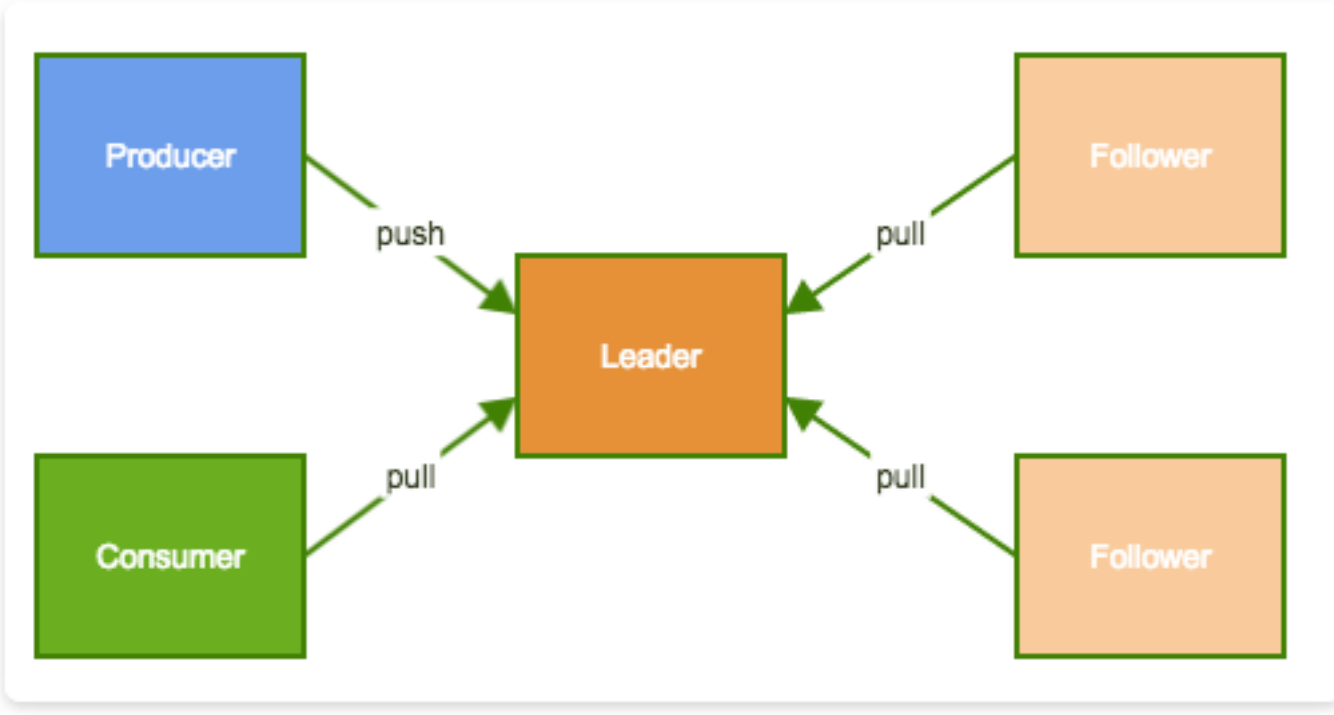
每个Partition分为多个Segment，每个Segment有.log和.index 两个文件，每个log文件承载具体的数据，每条消息都有一个递增的offset，Index文件是对log文件的索引，Consumer查找offset时使用的是二分法根据文件名去定位到哪个Segment，然后解析msg，匹配到对应的offset的msg。

2.1 Partition recovery过程

每个Partition会在磁盘记录一个RecoveryPoint,，记录已经flush到磁盘的最大offset。当broker 失败重启时，会进行loadLogs。首先会读取该Partition的RecoveryPoint，找到包含RecoveryPoint的segment及以后的segment， 这些segment就是可能没有完全flush到磁盘segments。然后调用segment的recover，重新读取各个segment的msg，并重建索引。每次重启KAFKA的broker时，都可以在输出的日志看到重建各个索引的过程。

2.2 数据同步

Producer和Consumer都只与Leader交互，每个Follower从Leader拉取数据进行同步。



Kafka数据同步


```
Topic:light-ubas-raw    PartitionCount:1        ReplicationFactor:2     Configs:retention.ms=2592000000
        Topic: light-ubas-raw   Partition: 0    Leader: 2       Replicas: 2,4   Isr: 2,4
```

Topic信息

如上图所示，ISR是所有不落后的replica集合，不落后有两层含义：距离上次FetchRequest的时间不大于某一个值或落后的消息数不大于某一个值，Leader失败后会从ISR中随机选取一个Follower做Leader，该过程对用户是透明的。

当Producer向Broker发送数据时,可以通过request.required.acks参数设置数据可靠性的级别。

此配置是表明当一次Producer请求被认为完成时的确认值。特别是，多少个其他brokers必须已经提交了数据到它们的log并且向它们的Leader确认了这些信息。

▪典型的值:
0： 表示Producer从来不等待来自broker的确认信息。这个选择提供了最小的时延但同时风险最大（因为当server宕机时，数据将会丢失）。
1：表示获得Leader replica已经接收了数据的确认信息。这个选择时延较小同时确保了server确认接收成功。
-1：Producer会获得所有同步replicas都收到数据的确认。同时时延最大，然而，这种方式并没有完全消除丢失消息的风险，因为同步replicas的数量可能是1。如果你想确保某些replicas接收到数据，那么你应该在Topic-level设置中

选项min.insync.replicas设置一下。

仅设置 acks= -1 也不能保证数据不丢失,当ISR列表中只有Leader时,同样有可能造成数据丢失。要保证数据不丢除了设置acks=-1，还要保证ISR的大小大于等于2。

▪具体参数设置：
request.required.acks:设置为-1 等待所有ISR列表中的Replica接收到消息后采算写成功。
min.insync.replicas: 设置为>=2,保证ISR中至少两个Replica。
Producer：要在吞吐率和数据可靠性之间做一个权衡。

KAFKA作为现代消息中间件中的佼佼者，以其速度和高可靠性赢得了广大市场和用户青睐，其中的很多设计理念都是非常值得我们学习的，本文所介绍的也只是冰山一角，希望能够对大家了解KAFKA有一定的作用。