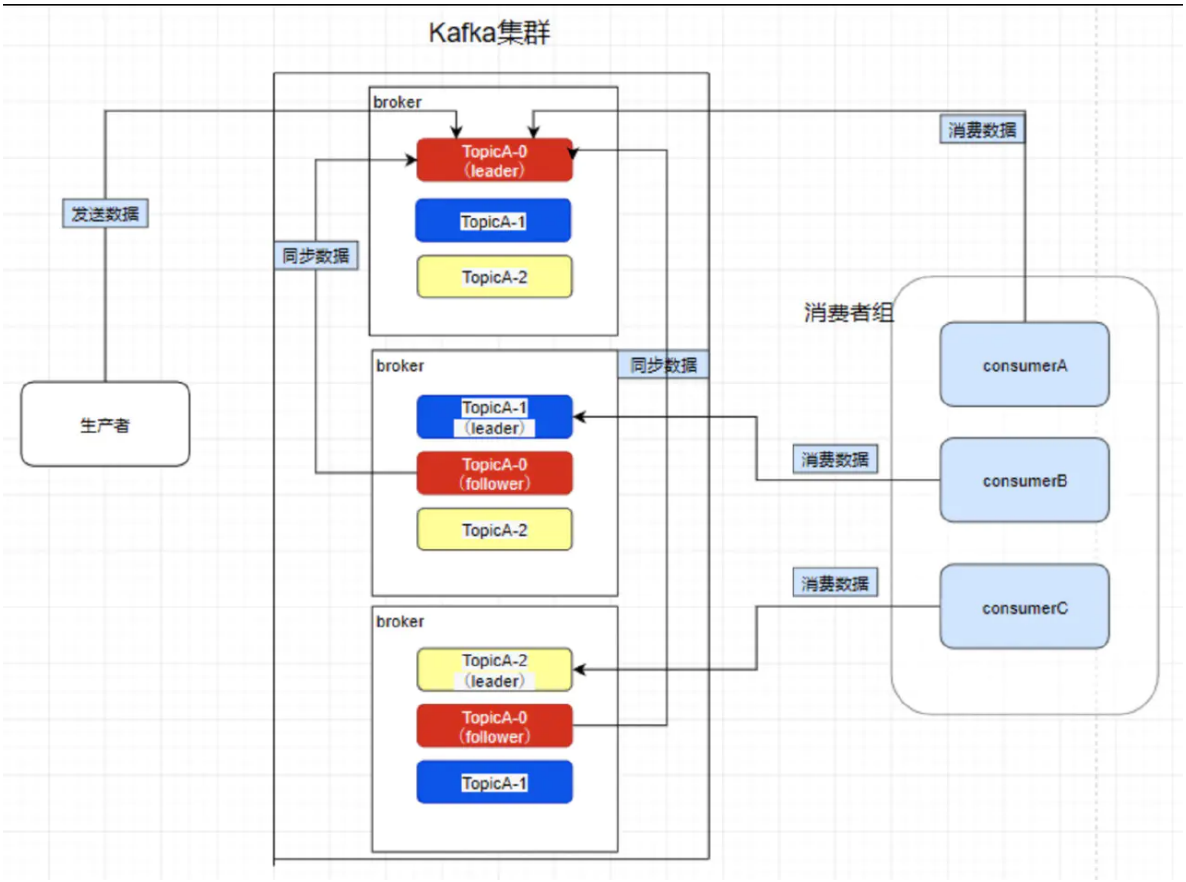
简介

> Apache Kafka是一个分布式发布 - 订阅消息系统和一个强大的队列, 可以处理大量的数据, 并使您能够将消息从一个端点传递到另一个端点. Kafka适合离线和在线消息消费. Kafka消息保留在磁盘上, 并在群集内复制以防止数据丢失. Kafka构建在ZooKeeper同步服务之上. 它与Apache Storm和Spark非常好地集成, 用于实时流式数据分析.
>
> Kafka 依赖于日志顺序写, 因此支持消息回溯和支撑高性能读写
>
> 依赖 Zookeeper

架构



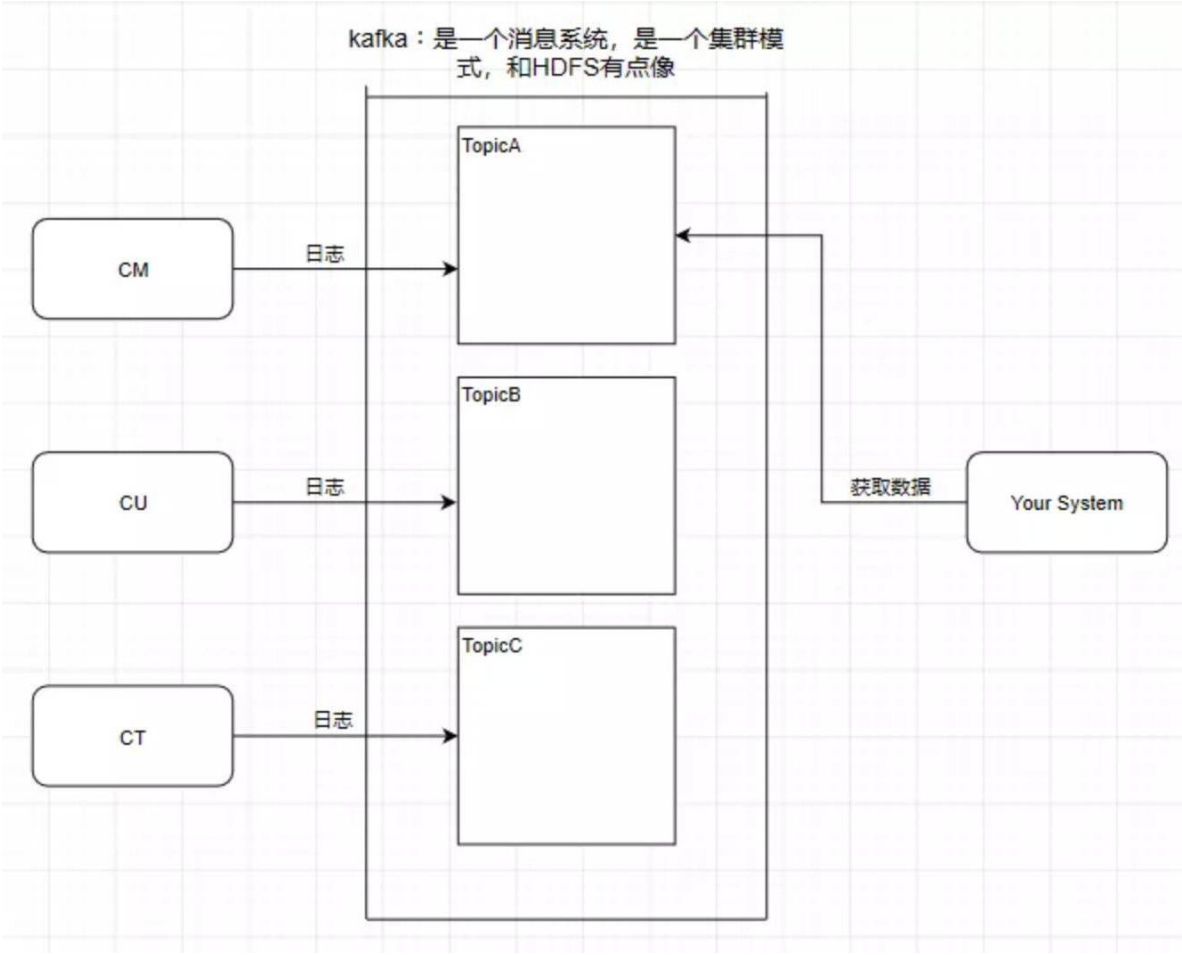
基础概念

Broker

> Server. 包含多个 Topic , Partition, 和 Replica. 负责协调 Producer 和 Consumer
>
> 主从结构为: 主节点为 Controller, 从节点为从节点 Kafka 启动是会往 Zookeeper 中注册当前 Broker 信息. 谁先注册谁就是 Controller. 读取注册上来的从节点的数据(通过监听机制), 生成集群的元数据信息, 之后把这些信息都分发给其他的服务器, 让其他服务器能感知到集群中其它成员的存在

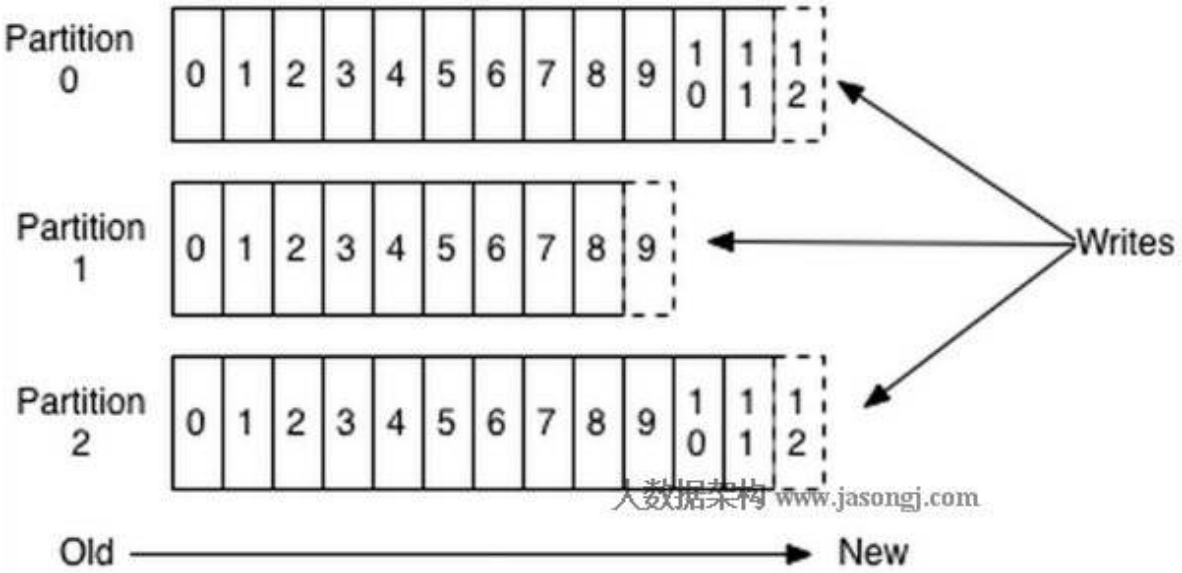
Topic

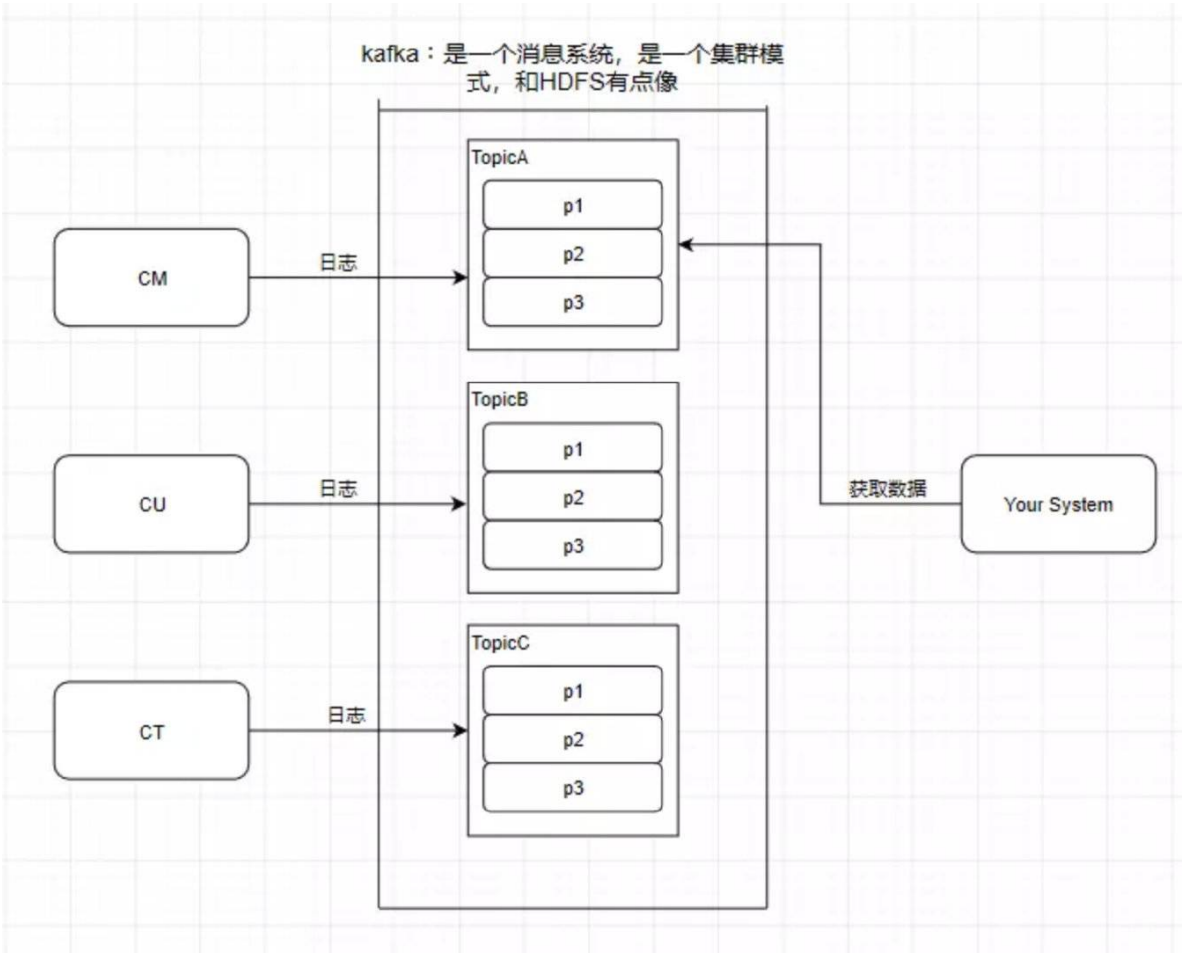
> 标准 MQ 中的 Queue. Kafka 中一个 Topic 的消息会保存在不同的 Partition (不同的 Broker)来保证高可用



Partition (分区)

> 可以理解为将标准 MQ 的 Queue 的消息进行拆分, 来实现高可用
>
> Producer 发送的 Message, 根据 key 和 partition 数进行 hash, 然后进行投递
>
> 一个分区只能被同一个 Consumer Group 中的一个 Consumer 消费. 分区内消费有序





Replica (备份)

> 每一个 Partition 的备份. Replica 的小于等于 Broker 的数量
>
> Leader: Replica领导节点, 每一个 Partition 都有对应的 Leader 节点(Broker). Producer 写数据时, 只会往 Leader 中写. Consumer 读数据也是从 Leader 中读 Follower: Replica跟随节点, 用于复制领导节点的数据. 复制 Leader 消息采用 pull (拉)模式

```
# Broker 设置副本数量 默认为 3
default.replication.factor

# Topic 设置副本数量
replication-factor
```

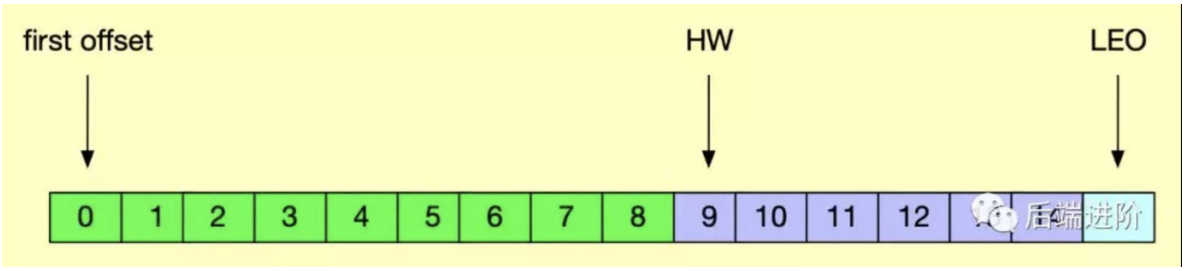
ISR (In-Sync Replica)

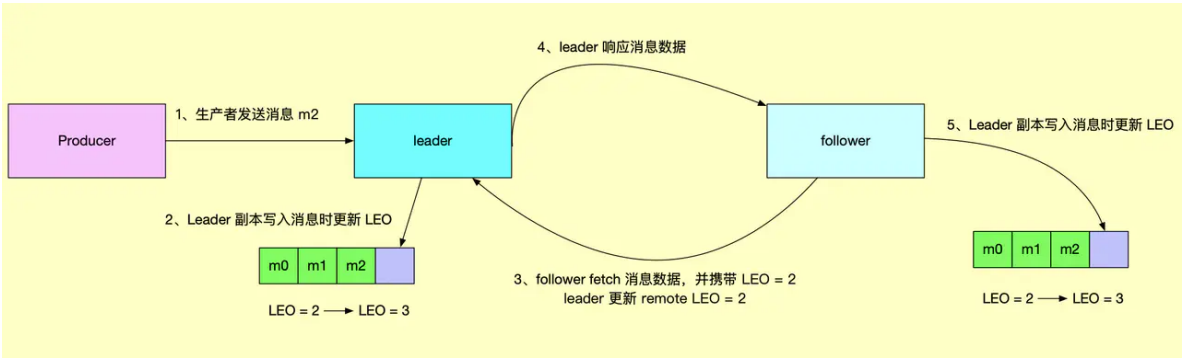
> Leader维护一个与其基本保持同步的Replica列表, 每个Partition都会有一个ISR, 而且是由leader动态维护. 如果一个flower比一个leader落后太多, 或者超过一定时间未发起数据复制请求, 则leader将其重ISR中移除. 当ISR中所有Replica都向Leader发送ACK时, leader才commit
>
> Leader 宕机之后, 会从 ISR 选择数据最新的 Follower 来当做 Leader 如果 ISR 全部宕机, 则选择第一个回复的 Replica 当做 Leader 节点 (消息可能会丢失或者重复消费)

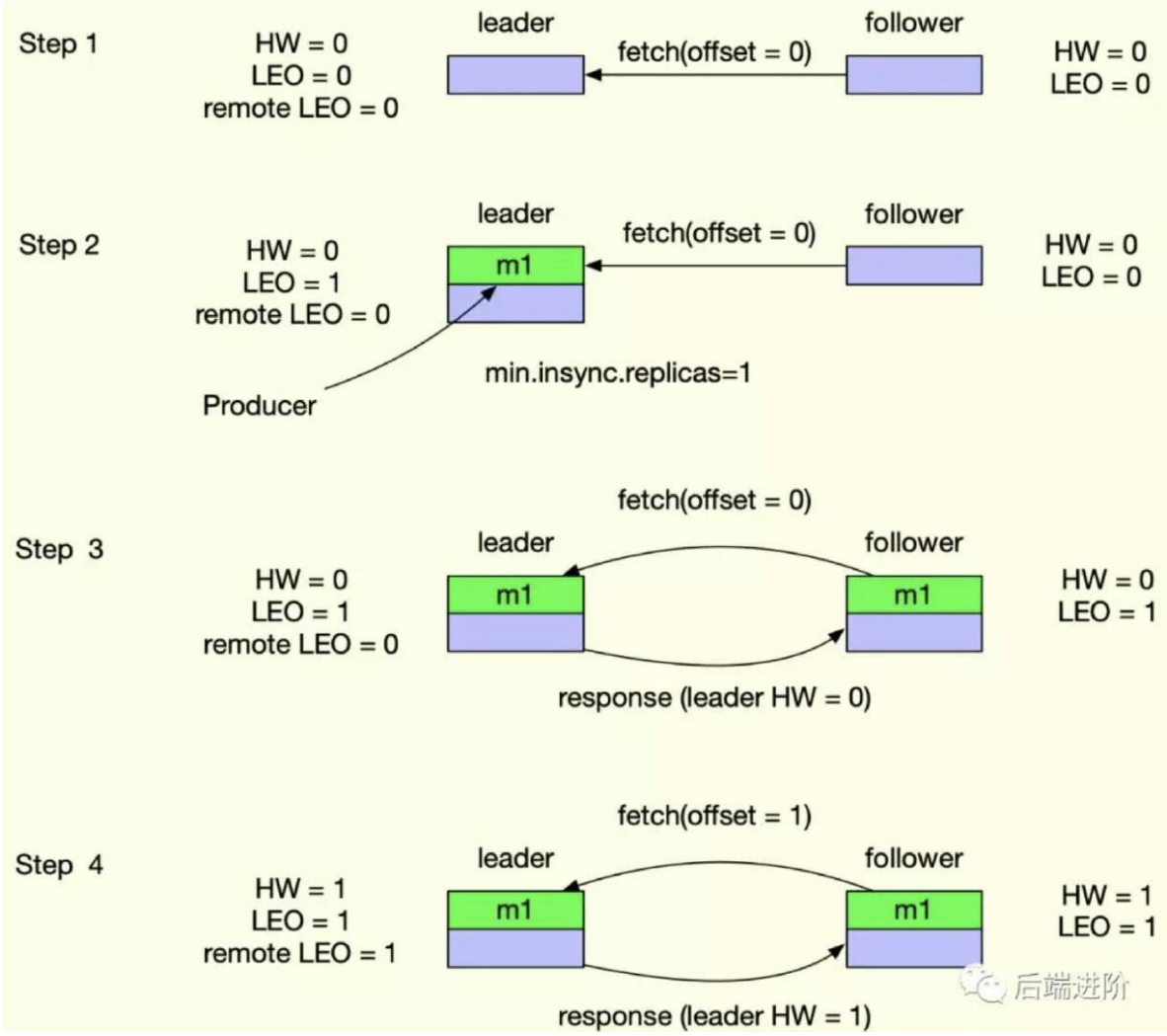
```
rerplica.lag.time.max.ms=10000
# 如果leader发现flower超过10秒没有向它发起fech请求，那么leader考虑这个flower是不是程序出了点问题
# 或者资源紧张调度不过来，它太慢了，不希望它拖慢后面的进度，就把它从ISR中移除.

rerplica.lag.max.messages=4000
# 相差4000条就移除
# flower慢的时候，保证高可用性，同时满足这两个条件后又加入ISR中，
# 在可用性与一致性做了动态平衡     亮点

min.insync.replicas=1
# 需要保证ISR中至少有多少个replica
```

水印备份机制

> LEO (last end offset): 日志末端位移, 记录了该副本对象底层日志文件中下一条消息的位移值, 副本写入消息的时候, 会自动更新 LEO 值 Leader 会保存两个 LEO 值, 一个是自己的 LEO 值, 另外一个是 remote 的 LEO 值. Follower 每次 fetch 请求都会携带当前 LEO, Leader 会选择最小的 LEO 来更新 HW
>
> HW (high watermark): 从名字可以知道, 该值叫高水印值, HW 一定不会大于 LEO 值, 小于 HW 值的消息被认为是"已提交"或"已备份"的消息, 并对消费者可见







Message

> 标准 MQ 的 Queue 中的 Message. 即一条消息

Producer

> 标准 MQ 中的发送方. 发送给 Broker 使用push (推)模式

数据一致性保证 (消息不丢失)

```
request.required.asks=0
# 0:相当于异步的，不需要leader给予回复，producer立即返回，发送就是成功，
那么发送消息网络超时或broker crash(1.Partition的Leader还没有commit消息 2.Leader与
Follower数据不同步)，既有可能丢失也可能会重发
# 1：当leader接收到消息之后发送ack，丢会重发，丢的概率很小
# -1：当所有的follower都同步消息成功后发送ack. 不会丢失消息
```

Consumer

> 标准 MQ 中的消费方. 接受 Broker 使用 pull (拉)模式, 默认 100ms 拉一次. Consumer 消费的是 Partition 的数据
>
> 消息丢失: 手动确认 ack 而不是自动提交 消息重复: 消费端幂等处理

Consumer Group

> 在 Kafka 中, 一个 Topic 是可以被一个消费组消费, 一个Topic 分发给 Consumer Group 中的 Consumer 进行消费, 保证同一条 Message 不会被不同的 Consumer 消费
>
> 注意: 当Consumer Group的 Consumer 数量大于 Partition 的数量时, 超过 Partition 的数量将会拿不到消息

分片规则

> Kafka分配Replica的算法有两种: RangeAssignor 和 RoundRobinAssignor
>
> 默认为RangeAssignor:
>
> 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) mod n)个Broker上

Rebalance (重平衡)

> Rebalance 本质上是一种协议, 规定了一个 Consumer Group 下的所有 consumer 如何达成一致, 来分配订阅 Topic 的每个分区
>
> Rebalance 发生时, 所有的 Consumer Group 都停止工作, 知道 Rebalance 完成

Coordinator

> Group Coordinator 是一个服务, 每个 Broker 在启动的时候都会启动一个该服务 Group Coordinator 的作用是用来存储 Group 的相关 Meta 信息, 并将对应 Partition 的 Offset 信息记录到 Kafka 内置 Topic(__consumer_offsets)中 Kafka 在0.9之前是基于 Zookeeper 来存储Partition 的 offset 信息(consumers/{group}/offsets/{topic}/{partition}), 因为 Zookeeper 并不适用于频繁的写操作, 所以在0.9之后通过内置 Topic 的方式来记录对应 Partition 的 offset

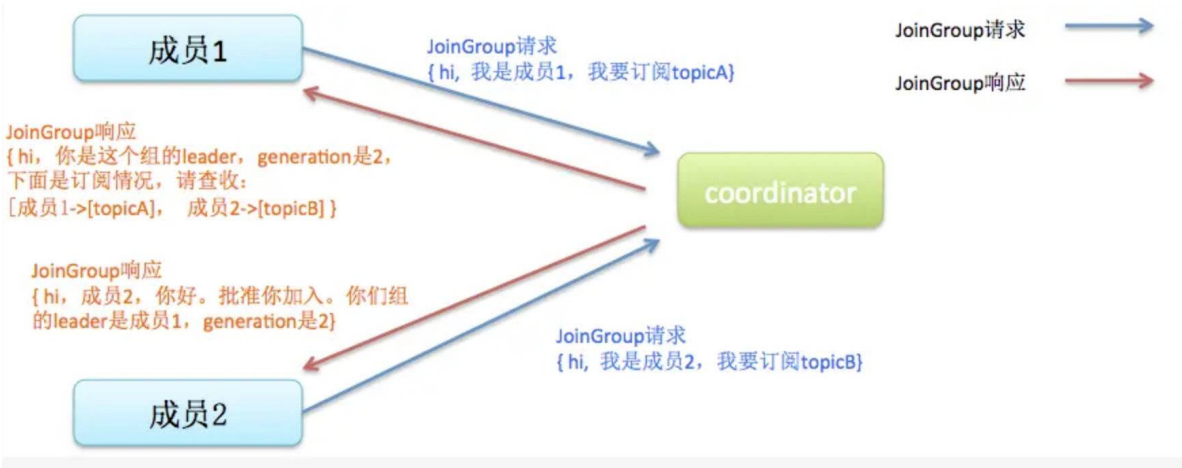
触发条件

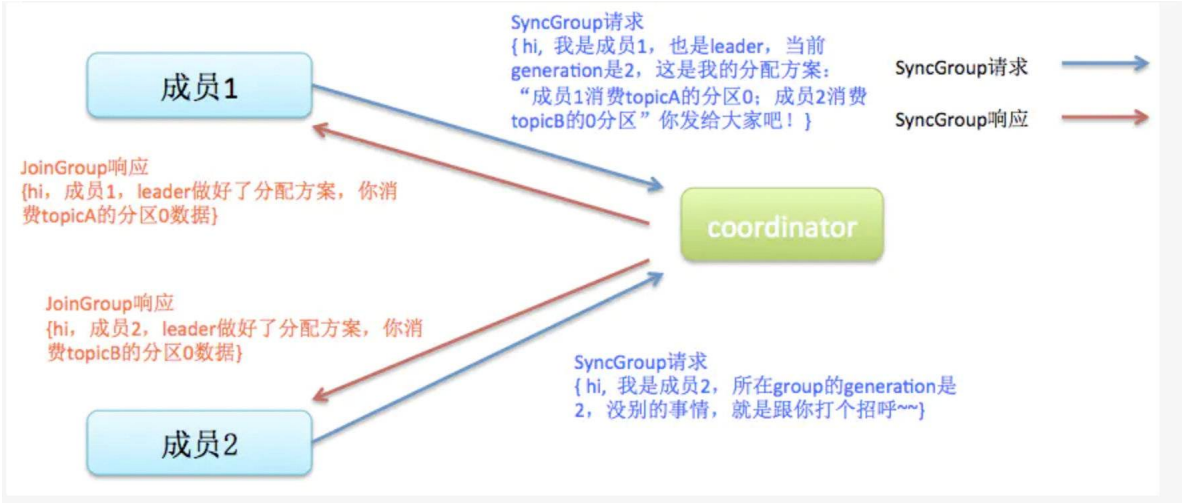
> 1. 组成员个数发生变化
>   1. 新的消费者加入到消费组
>   2. 消费者主动退出消费组
>   3. 消费者被动下线. 比如消费者长时间的GC, 网络延迟导致消费者长时间未向Group Coordinator发送心跳请求, 均会认为该消费者已经下线并踢出
> 2. 订阅的 Topic 的 Consumer Group 个数发生变化
> 3. Topic 的分区数发生变化

Rebalace 流程

> Rebalance 过程分为两步：Join 和 Sync
>
> 1. Join: 顾名思义就是加入组. 这一步中, 所有成员都向 Coordinator 发送 JoinGroup 请求, 请求加入消费组. 一旦所有成员都发送了 JoinGroup 请求, Coordinator 会从中选择一个 Consumer 担任 Leader 的角色, 并把组成员信息以及订阅信息发给 Consumer Leader 注意 Consumer Leader 和 Coordinator不是一个概念. Consumer Leader负责消费分配方案的制定
> 2. Sync: Consumer Leader 开始分配消费方案, 即哪个 Consumer 负责消费哪些 Topic 的哪些 Partition. 一旦完成分配, Leader 会将这个方案封装进 SyncGroup 请求中发给 Coordinator, 非 Leader 也会发 SyncGroup 请求, 只是内容为空. Coordinator 接收到分配方案之后会把方案塞进SyncGroup的Response中发给各个Consumer. 这样组内的所有成员就都知道自己应该消费哪些分区了

> 各个场景参考: [Kafka Rebalance机制分析]





如何避免 Rebalance

> 对于触发条件的 2 和 3, 我们可以人为避免. 1 中的 1 和 3 人为也可以尽量避免, 主要核心为 3

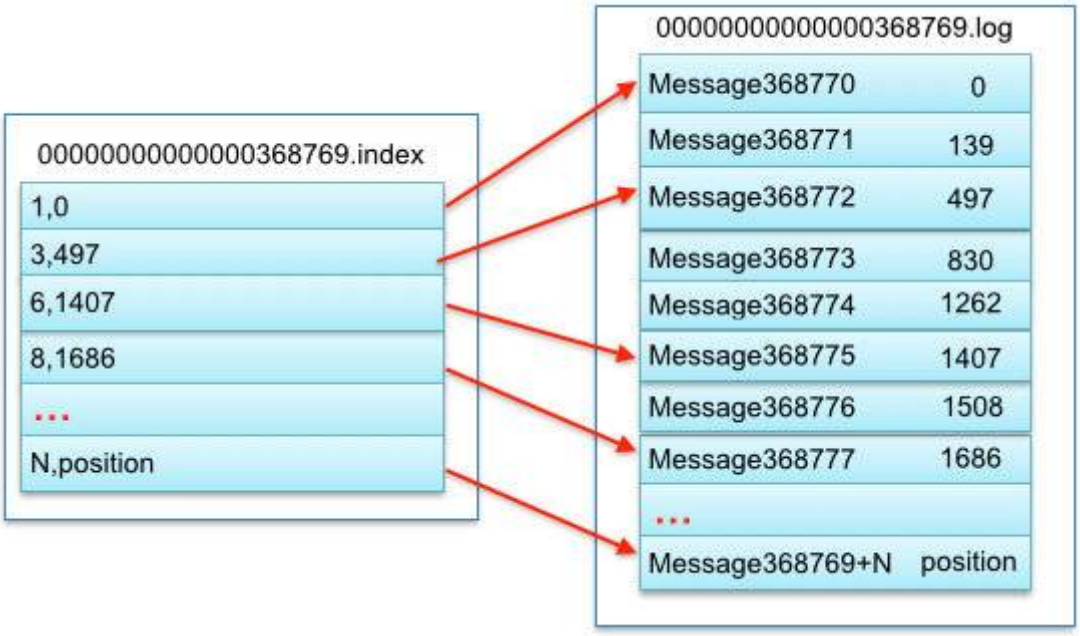
```
# 心跳相关
session.timeout.ms = 6s
heartbeat.interval.ms = 2s

# 消费时间
max.poll.interval.ms
```

日志索引

> Kafka 能支撑 TB 级别数据, 在日志级别有两个原因: 顺序写和日志索引. 顺序写后续会讲
>
> Kafka 在一个日志文件达到一定数据量 (1G) 之后, 会生成新的日志文件, 大数据情况下会有多个日志文件, 通过偏移量来确定到某行纪录时, 如果遍历所有的日志文件, 那效率自然是很差的. Kafka 在日志级别上抽出来一层日志索引, 来方便根据 offset 快速定位到是某个日志文件
>
> 每一个 partition 对应多个个 log 文件(最大 1G), 每一个 log 文件又对应一个 index 文件
>
> 通过 offset 查找 Message 流程:
>
> 1. 先根据 offset (例: 368773), 二分定位到最大 小于等于该 offset 的 index 文件 (368769.index)

2. 通过二分(368773 - 368769 = 4)定位到 index 文件 (368769.index) 中最大 小于等于该 offset 的 对于的 log 文件偏移量(3, 497)
3. 通过定位到该文件的消息行(3, 497), 然后在往后一行一行匹配揭露(368773 830)

```
00000000000000000000.index
00000000000000000000.log
00000000000000368769.index
00000000000000368769.log
00000000000000737337.index
00000000000000737337.log
00000000000001105814.index
00000000000001105814.log
.....................
```



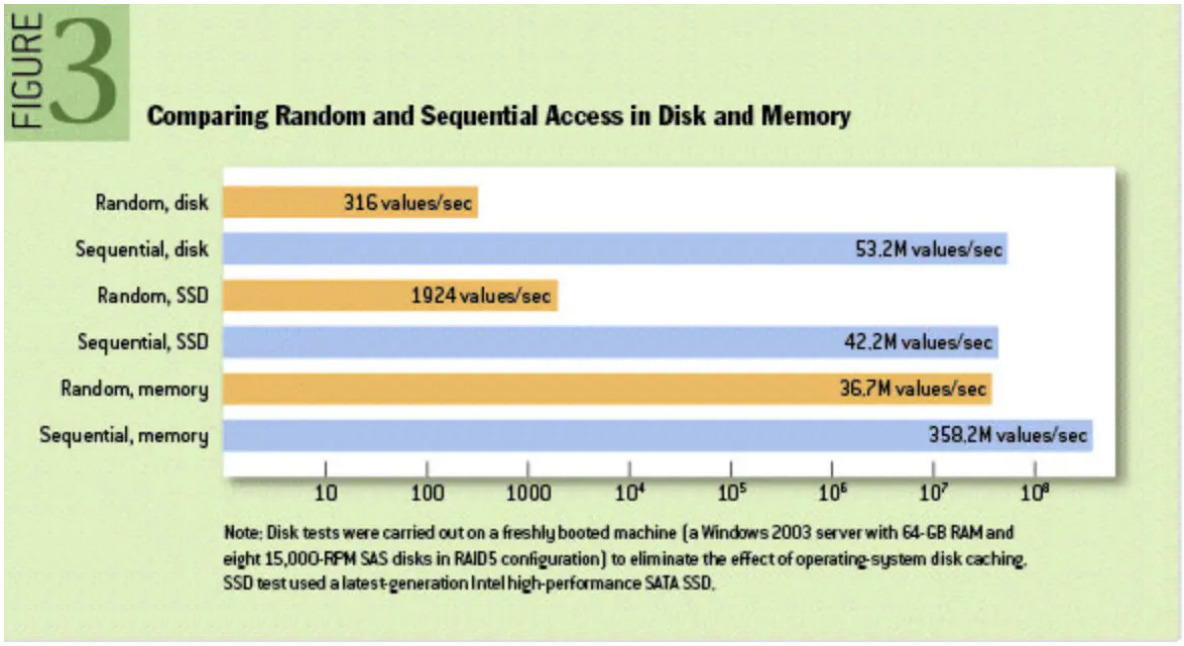
高性能, 高吞吐

分区的原因

> 如果我们假设像标准 MQ 的 Queue, 为了保证一个消息只会被一个消费者消费, 那么我们第一想到的就是加锁. 对于发送者, 在多线程并且非顺序写环境下, 保证数据一致性, 我们同样也要加锁. 一旦考虑到加锁, 就会极大的影响性能. 我们再来看Kafka 的 Partition, Kafka 的消费模式和发送模式都是以 Partition 为分界. 也就是说对于一个 Topic 的并发量限制在于有多少个 Partition, 就能支撑多少的并发. 可以参考 Java 1.7 的 ConcurrentHashMap 的桶设计, 原理一样, 有多少桶, 支持多少的并发

顺序写

> 从图中可以看出来, 磁盘的顺序写的性能要比内存随机写的还要强. 磁盘顺序写和随机写的差距也是天壤之别



批发送

> 批处理是一种常用的用于提高I/O性能的方式. 对Kafka而言, 批处理既减少了网络传输的Overhead, 又提高了写磁盘的效率. Kafka 0.82 之后是将多个消息合并之后再发送, 而并不是send一条就立马发送(之前支持)

```
# 批量发送的基本单位, 默认是16384Bytes, 即16kB
batch.size

# 延迟时间
linger.ms

# 两者满足其一便发送
```

数据压缩

> 数据压缩的一个基本原理是, 重复数据越多压缩效果越好. 因此将整个Batch的数据一起压缩能更大幅度减小数据量, 从而更大程度提高网络传输效率

> Broker接收消息后，并不直接解压缩，而是直接将消息以压缩后的形式持久化到磁盘 Consumer接受到压缩后的数据再解压缩
>
> 整体来讲: Producer 到 Broker, 副本复制, Broker 到 Consumer 的数据都是压缩后的数据, 保证高效率的传输

Page Cache & MMap

> 利用 MMap 的读写文件也会依赖于 Page Cache, Page Cache 可以单独存在, 但是 MMap 会依赖于 Page Cache. MMap 是将文件映射到内存中, 底层来讲就是将 Page Cache 映射到用户空间, 从而使得用户空间可以直接操作文件

Page Cache

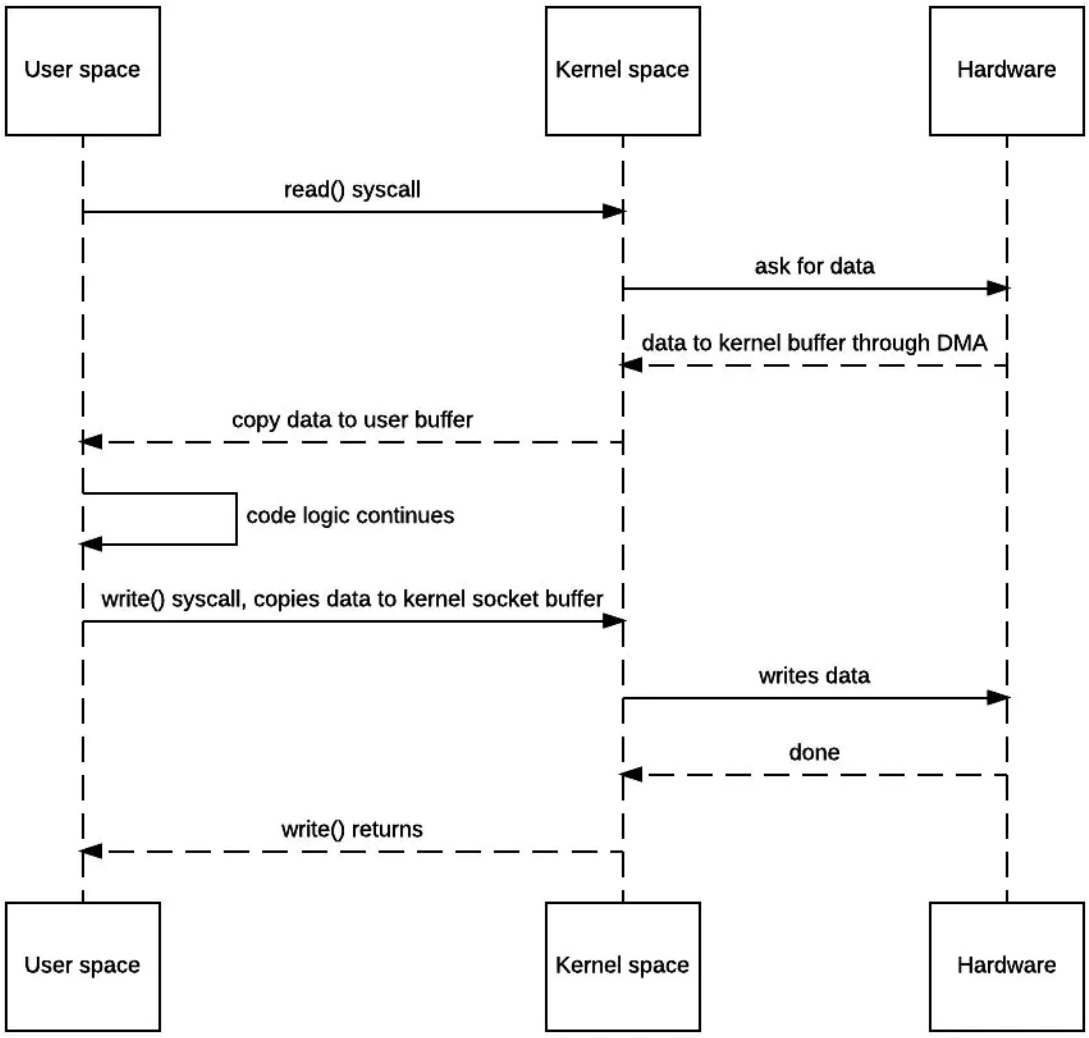
> 内核会为每个文件单独维护一个page cache, 用户进程对于文件的大多数读写操作会直接作用到page cache上, 内核会选择在适当的时候将page cache中的内容写到磁盘上 (当然我们可以手工fsync控制回写), 这样可以大大减少磁盘的访问次数, 从而提高性能
>
> 至于刷盘时机: page cache中的数据会随着内核中flusher线程的调度以及对sync()/fsync()的调用写回到磁盘, 同样也可以通过手动调用写磁盘

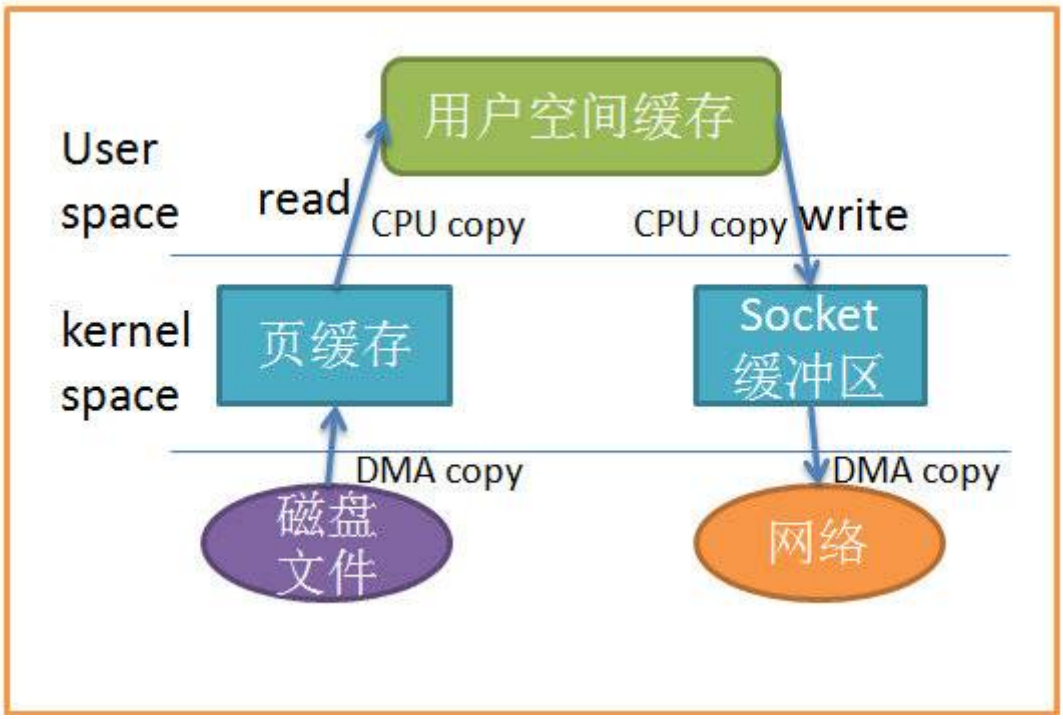
MMap (Memory Mapped Files, 内存映射文件)

> MMap 是将一个文件或者其它对象映射进内存. Java 支持的零拷贝参考MappedByteBuffer

正常的写流程:

> 1. JVM向OS发出read()系统调用, 触发上下文切换, 从用户态切换到内核态
> 2. 从外部存储 (如硬盘)读取文件内容通过直接内存访问(DMA)存入内核地址空间的缓冲区
> 3. 将数据从内核缓冲区拷贝到用户空间缓冲区, read()系统调用返回, 并从内核态切换回用户态
> 4. JVM向OS发出write()系统调用, 触发上下文切换, 从用户态切换到内核态
> 5. 将数据从用户缓冲区拷贝到内核中与目的地Socket关联的缓冲区
> 6. 数据最终经由Socket通过DMA传送到硬件(如网卡)缓冲区, write()系统调用返回, 并从内核态切换回用户态
>
> 读取: 用户态切换到内核态, 在内核空间将文件读取到缓冲区, 再从内核区拷贝到用户缓冲区, 最后切换回用户态 写入: 用户态切换到内核态, 将数据从用户缓冲区拷贝到内核缓冲区, 再由DMA发送到硬件缓冲区, 最后切换回用户态
>
> 我们发现, 其实用户空间和内核空间的切换一共四次, 文件拷贝四次. 如果有一种方式文件

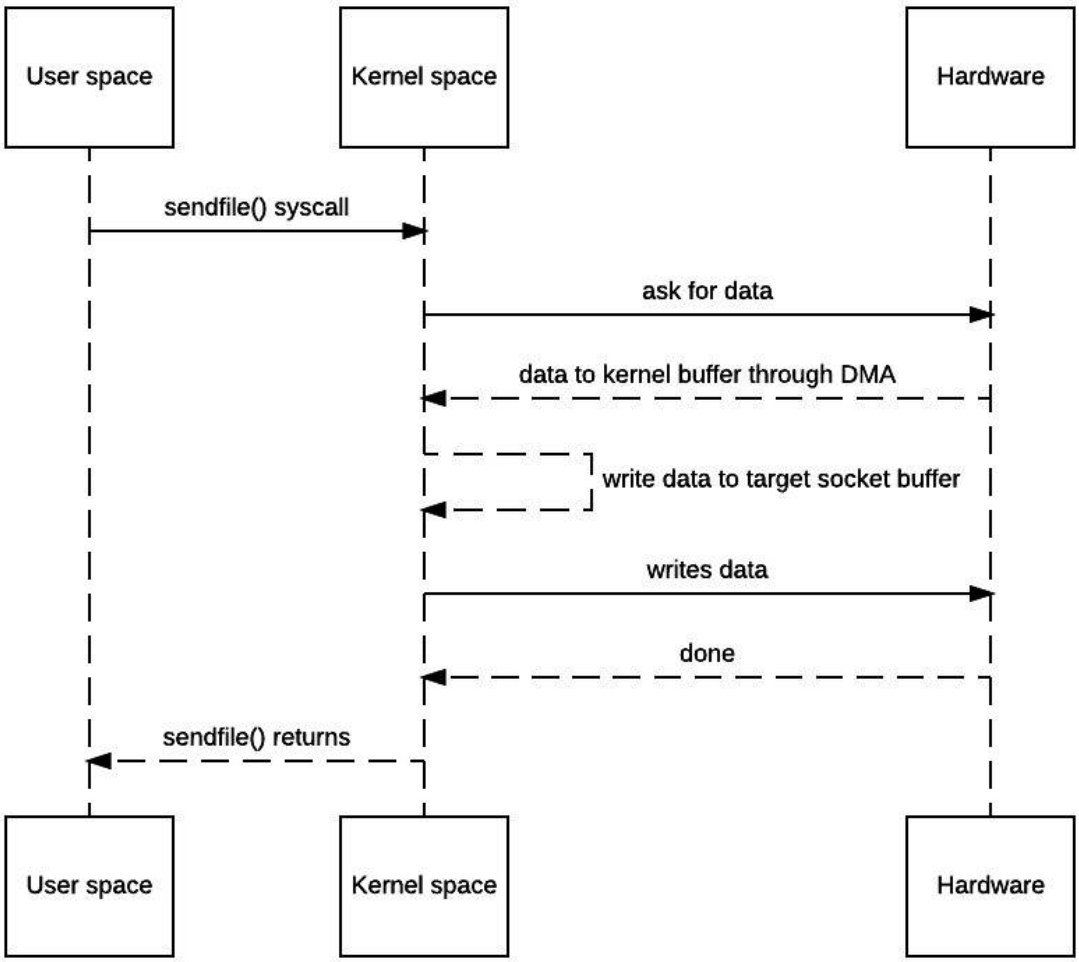


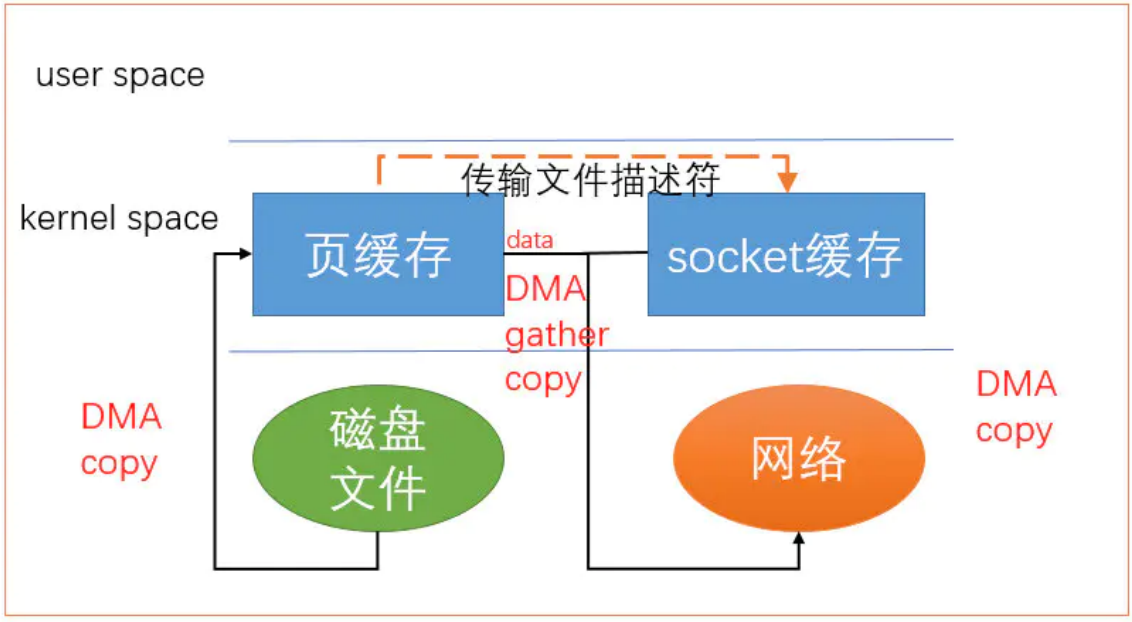


mmap 的写流程

> 1. JVM向OS发出sendfile()系统调用
> 2. 从外部存储 (如硬盘)读取文件内容通过直接内存访问(DMA)存入内核地址空间的缓冲区

> 3. 数据最终经由Socket通过DMA传送到硬件(如网卡)缓冲区, write()系统调用返回, 并从内核态切换回用户态
>
> 通过零拷贝, 我们减少了两次用户空间和内核空间的拷贝和一次内核空间和 Socket 缓冲区的拷贝, 总共节省三次拷贝, 并且没有用户态和内核态的转换





Kafka 的运用

> Kafka 写数据的时候, 会直接写到 Page Cache 中, 消费者拉数据时也会经过 Page Cache. 如果 Kafka 的写速率和消费速率差不多, 那么整个生产和消费过程是不会经过磁盘 IO. 全部都是内存操作. 对于 Page Cache 没有刷盘导致的数据丢失, 如果发送方配置消息至少有一个副本接受, 那么也只会有已一次同步刷盘
>
> Kafka为什么不自己管理缓存, 而非要用page cache?
>
> 1. JVM中一切皆对象, 数据的对象存储会带来所谓 object overhead 浪费空间
> 2. 如果由JVM来管理缓存, 会受到GC的影响, 并且过大的堆也会拖累GC的效率, 降低吞吐量
> 3. 一旦程序崩溃, 自己管理的缓存数据会全部丢失