

# 分布式消息队列

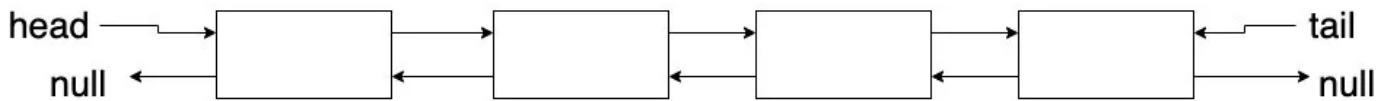
[https://mp.weixin.qq.com/s/-MXA4T-ei\\_U5ewXUNZ0QdQ](https://mp.weixin.qq.com/s/-MXA4T-ei_U5ewXUNZ0QdQ)

## 一、消息队列的演进

分布式消息队列中间件是大型分布式系统中常见的中间件。消息队列主要解决应用耦合、异步消息、流量削锋等问题，具有高性能、高可用、可伸缩和最终一致性等特点。消息队列已经逐渐成为企业应用系统内部通信的核心手段，使用较多的消息队列有 RabbitMQ、RocketMQ、ActiveMQ、Kafka、ZeroMQ、Pulsar 等，此外，利用数据库（如 Redis、MySQL 等）也可实现消息队列的部分基本功能。

### 1. 基于 OS 的 MQ

单机消息队列可以通过操作系统原生的进程间通信机制来实现，如消息队列、共享内存等。比如我们可以在共享内存中维护一个双端队列：



消息产出进程不停地往队列里添加消息，同时消息消费进程不断地从队尾有序地取出这些消息。添加消息的任务我们称为 producer，而取出并使用消息的任务，我们称之为 consumer。这种模式在早期单机多进程模式中比较常见，比如 IO 进程把收到的网络请求存入本机 MQ，任务处理进程从本机 MQ 中读取任务并进行处理。

单机 MQ 易于实现，但是缺点也很明显：因为依赖于单机 OS 的 IPC 机制，所以无法实现分布式的消息传递，并且消息队列的容量也受限于单机资源。

### 2. 基于 DB 的 MQ

即使用存储组件（如 Mysql、Redis 等）存储消息，然后在消息的生产侧和消费侧实现消息的生产消费逻辑，从而实现 MQ 功能。以 Redis 为例，可以使用 Redis 自带的 list 实现。Redis list 使用 lpush 命令，从队列左边插入数据；使用 rpop 命令，从队列右边取出数据。与单机 MQ 相比，该方案至少满足了分布式，但是仍然带有很多无法接受的缺陷。

- 热 key 性能问题：不论是用 codis 还是 twemproxy 这种集群方案，对某个队列的读写请求最终都会落到同一台 redis 实例上，并且无法通过扩容来解决问题。如果对某个 list 的并发读写非常高，就产生了无法解决的热 key，严重可能导致系统崩溃
- 没有消费确认机制：每当执行 rpop 消费一条数据，那条消息就被从 list 中永久删除了。如果消费者消费失败，这条消息也没法找回了。
- 不支持多订阅者：一条消息只能被一个消费者消费，rpop 之后就没了。如果队列中存储的是应用的日志，对于同一条消息，监控系统需要消费它来进行可能的报警，BI 系统需要消费它来绘制报表，链路追踪需要消费它来绘制调用关系……这种场景 redis list 就没办法支持了
- 不支持二次消费：一条消息 rpop 之后就没了。如果消费者程序运行到一半发现代码有 bug，修复之后想从头再消费一次就不行了。

针对上述缺点，redis 5.0 开始引入 stream 数据类型，它是专门设计成为消息队列的数据结构，借鉴了很多 kafka 的设计，但是随着很多分布式 MQ 组件的出现，仍然显得不够友好，毕竟 Redis 天生就不是用来做消息转发的。

### 3. 专用分布式 MQ 中间件

随着时代的发展，一个真正的消息队列，已经不仅仅是一个队列那么简单了，业务对 MQ 的吞吐量、扩展性、稳定性、可靠性等都提出了严苛的要求。因此，专用的分布式消息中间件开始大量出现。常见的有 RabbitMQ、RocketMQ、ActiveMQ、Kafka、ZeroMQ、Pulsar 等等。

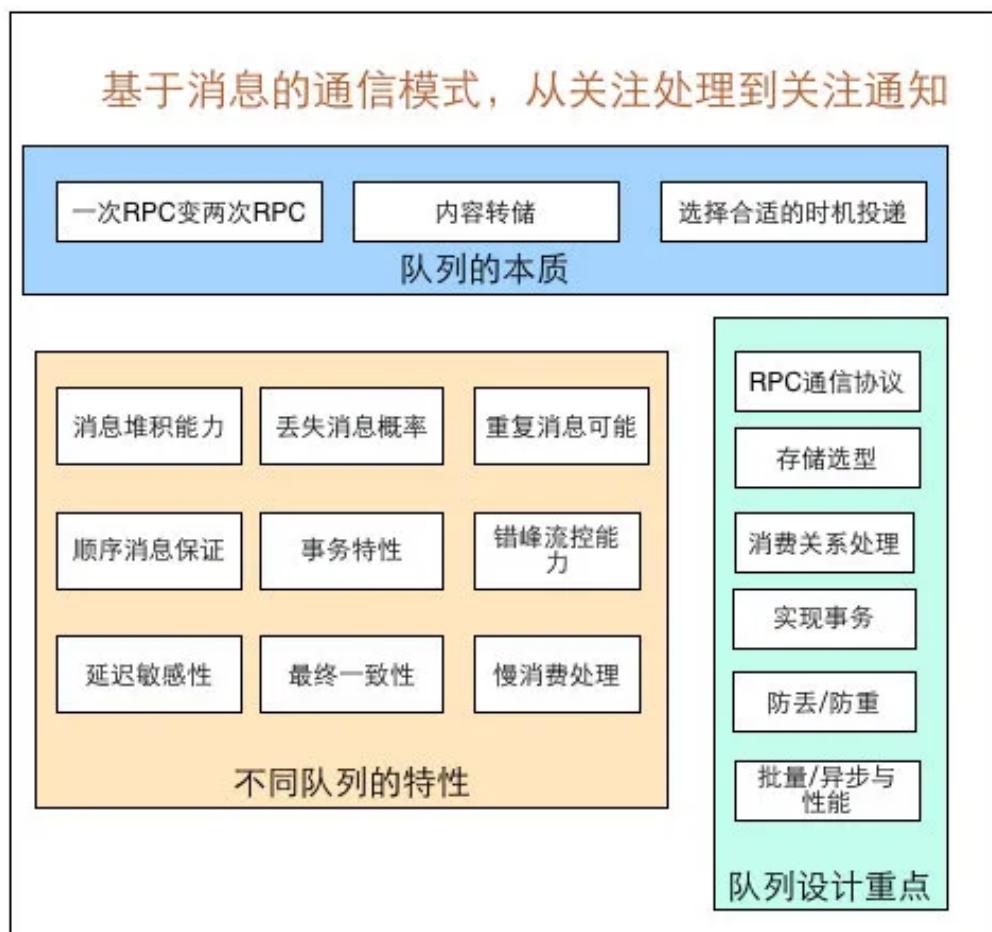
## 二、消息队列设计要点

消息队列本质上是一个消息的转发系统，把一次 RPC 就可以直接完成的消息投递，转换成多次 RPC 间接完成，这其中包含两个关键环节：

1. 消息转储；
2. 消息投递：时机和对象；

基于此，消息队列的整体设计思路是：

- 确定整体的数据流向：如 producer 发送给 MQ，MQ 转发给 consumer，consumer 回复消费确认，消息删除、消息备份等。
- 利用 RPC 将数据流串起来，最好基于现有的 RPC 框架，尽量做到无状态，方便水平扩展。
- 存储选型，综合考虑性能、可靠性和开发维护成本等诸多因素。
- 消息投递，消费模式 push、pull。
- 消费关系维护，单播、多播等，可以利用 zk、config server 等保存消费关系。
- 高级特性，如可靠投递，重复消息，顺序消息等，很多高级特性之间是相互制约的关系，这里要充分结合应用场景做出取舍。



## 1.MQ 基本特性

### RPC 通信

MQ 组件要实现和生产者以及消费者进行通信功能，这里涉及到 RPC 通信问题。消息队列的 RPC，和普通的 RPC 没有本质区别。对于负载均衡、服务发现、序列化协议等等问题都可以借助现有 RPC 框架来实现，避免重复造轮子。

### \*存储系统\*

存储可以做成很多方式。比如存储在内存里，存储在分布式 KV 里，存储在磁盘里，存储在数据库里等等。但归结起来，主要有持久化和非持久化两种。

持久化的形式能更大程度地保证消息的可靠性（如断电等不可抗外力），并且理论上能承载更大限度的消息堆积（外存的空间远大于内存）。但并不是每种消息都需要持久化存储。很多消息对于投递性能的要求大于可靠性的要求，且数量极大（如日志）。这时候，消息不落地直接暂存内存，尝试几次 failover，最终投递出去也未尝不可。常见的消息队列普遍两种形式都支持。

从速度来看，理论上，文件系统>分布式 KV（持久化）>分布式文件系统>数据库，而可靠性却相反。还是要从支持的业务场景出发作出最合理的选择。

### \*高可用\*

MQ 的高可用，依赖于 RPC 和存储的高可用。通常 RPC 服务自身都具有服务自动发现，负载均衡等功能，保证了其高可用。存储的高可用，例如 Kafka，使用分区加主备模式，保证每一个分区内的高可用性，也就是每一个分区至少要有一个备份且需要做数据的同步。

### \*推拉模型\*

push 和 pull 模型各有利弊，两种模式也都有被市面上成熟的消息中间件选用。

#### 1.慢消费

慢消费是 push 模型最大的致命伤，如果消费者的速度比发送者的速度慢很多，会出现两种恶劣的情况：

1.消息在 broker 的堆积。假设这些消息都是有用的无法丢弃的，消息就要一直在 broker 端保存。

2.broker 推送给 consumer 的消息 consumer 无法处理，此时 consumer 只能拒绝或者返回错误。

而 pull 模式下，consumer 可以按需消费，不用担心自己处理不了的消息来骚扰自己，而 broker 堆积消息也会相对简单，无需记录每一个要发送消息的状态，只需要维护所有消息的队列和偏移量就可以了。所以对于慢消费，消息量有限且到来的速度不均匀的情况，pull 模式比较合适。

#### 2.消息延迟与忙等

这是 pull 模式最大的短板。由于主动权在消费方，消费方无法准确地决定何时去拉取最新的消息。如果一次 pull 取到消息了还可以继续去 pull，如果没有 pull 取到则需要等待一段时间重新 pull。

### \*消息投放时机\*

即消费者应该在什么时机消费消息。一般有以下三种方式：

1. 攒够了一定数量才投放。
2. 到达了一定时间就投放。
3. 有新的数据到来就投放。

至于如何选择，也要结合具体的业务场景来决定。比如，对及时性要求高的数据，可用采用方式 3 来完成。

#### \*消息投放对象\*

不管是 JMS 规范中的 Topic/Queue，Kafka 里面的 Topic/Partition/ConsumerGroup，还是 AMQP（如 RabbitMQ）的 Exchange 等等，都是为了维护消息的消费关系而抽象出来的概念。本质上，消息的消费无外乎点到点的一对一单播，或一对多广播。另外比较特殊的情况是组间广播、组内单播。比较通用的设计是，不同的组注册不同的订阅，支持组间广播。组内不同的机器，如果注册一个相同的 ID，则单播；如果注册不同的 ID(如 IP 地址+端口)，则广播。

例如 pulsar 支持的订阅模型有：

- Exclusive：独占型，一个订阅只能有一个消费者消费消息。
- Failover：灾备型，一个订阅同时只有一个消费者，可以有多个备份消费者。一旦主消费者故障则备份消费者接管。不会出现同时有两个活跃的消费者。
- Shared：共享型，一个订阅中同时可以有多个消费者，多个消费者共享 Topic 中的消息。
- Key\_Shared：键共享型，多个消费者各取一部分消息。

通常会在公共存储上维护广播关系，如 config server、zookeeper 等。

## 2.队列高级特性

常见的高级特性有可靠投递、消息丢失、消息重复、事务等等，他们并非是 MQ 必备的特性。由于这些特性可能是相互制约的，所以不可能完全兼顾。所以要依照业务的需求，来仔细衡量各种特性实现的成本、利弊，最终做出最为合理的设计。

#### \*可靠投递\*

如何保证消息完全不丢失？

直观的方案是，在任何不可靠操作之前，先将消息落地，然后操作。当失败或者不知道结果（比如超时）时，消息状态是待发送，定时任务不停轮询所有待发送消息，最终一定可以送达。但是，这样必然导致消息可能会重复，并且在异常情况下，消息延迟较大。

例如：

- producer 往 broker 发送消息之前，需要做一次落地。
- 请求到 server 后，server 确保数据落地后再告诉客户端发送成功。
- 支持广播的消息队列需要对每个接收者，持久化一个发送状态，直到所有接收者都确认收到，才可删除消息。

即对于任何不能确认消息已送达的情况，都要重推消息。但是，随着而来的问题就是消息重复。在消息重复和消息丢失之间，无法兼顾，要结合应用场景做出取舍。

#### \*消费确认\*

当 broker 把消息投递给消费者后，消费者可以立即确认收到了消息。但是，有些情况消费者可能需要再次接收该消息（比如收到消息、但是处理失败），即消费者主动要求重发消息。所以，要允许消费者主动进行消费确认。

#### \*顺序消息\*

对于 push 模式，要求支持分区且单分区只支持一个消费者消费，并且消费者只有确认一个消息消费后才能 push 另外一个消息，还要发送者保证发送顺序唯一。

对于 pull 模式，比如 kafka 的做法：

1. producer 对应 partition，并且单线程。
2. consumer 对应 partition，消费确认（或批量确认），单线程消费。

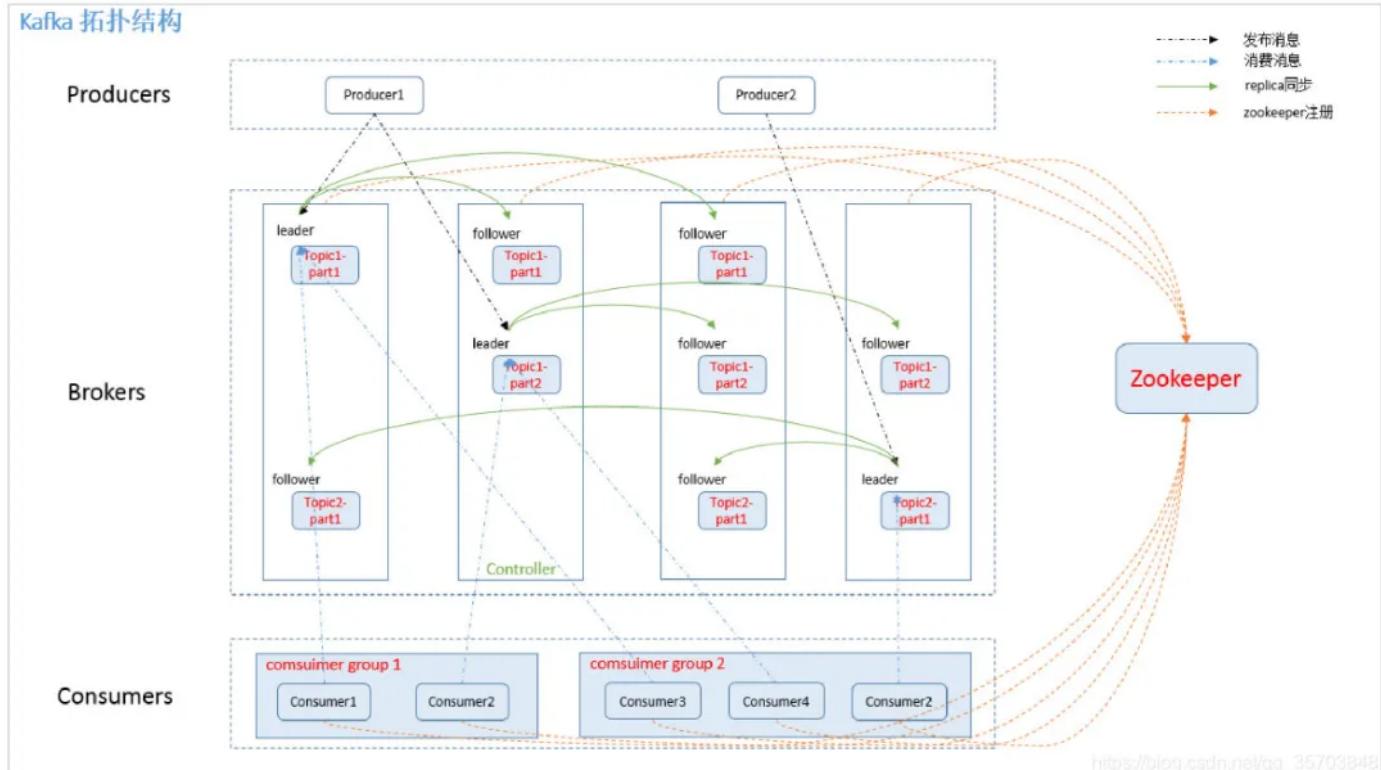
但是这样也只是实现了消息的分区有序性，并不一定全局有序。总体而言，要求消息有序的 MQ 场景还是比较少的。

## 三、Kafka

Kafka 是一个分布式发布订阅消息系统。它以高吞吐、可持久化、可水平扩展、支持流数据处理等多种特性而被广泛使用（如 Storm、Spark、Flink）。在大数据系统中，数据需要在各个子系统中高性能、低延迟的不停流转。传统的企业消息系统并不是非常适合大规模的数据处理，但 Kafka 出现了，它可以高效的处理实时消息和离线消息，降低编程复杂度，使得各个子系统可以快速高效的进行数据流转，Kafka 承担高速数据总线的作用。

### kafka 基础概念

- **Broker** Kafka 集群包含一个或多个服务器，这种服务器被称为 broker。
- **Topic** Topic 在逻辑上可以被认为是一个 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 的吞吐率可以线性提高，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。
- **Partition** Partition 是物理上的概念，每个 Topic 包含一个或多个 Partition。
- **Producer** 负责发布消息到 Kafka broker。
- **Consumer** 消息消费者，向 Kafka broker 读取消息的客户端。
- **Consumer Group** 每个 Consumer 属于一个特定的 Consumer Group（可为每个 Consumer 指定 group name，若不指定 group name 则属于默认的 group）。



kafka实现原理6

一个典型的 kafka 集群包含若干 Producer, 若干个 Broker (kafka 支持水平扩展) 、若干个 Consumer Group, 以及一个 zookeeper 集群。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。多个 broker 协同工作, producer 和 consumer 部署在各个业务逻辑中。kafka 通过 zookeeper 管理集群配置及服务协同。

这样就组成了一个高性能的分布式消息发布和订阅系统。Kafka 有一个细节是和其他 mq 中间件不同的点, producer 发送消息到 broker 的过程是 push, 而 consumer 从 broker 消费消息的过程是 pull, 主动去拉数据。而不是 broker 把数据主动发送给 consumer。

Producer 发送消息到 broker 时, 会根据 Partition 机制选择将其存储到哪一个 Partition。如果 Partition 机制设置合理, 所有消息可以均匀分布到不同的 Partition 里, 这样就实现了负载均衡。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, 而有了 Partition 后, 不同的消息可以并行写入不同 broker 的不同 Partition 里, 极大的提高了吞吐率。

## Kafka 特点

### \*优点：\*

- 高性能: 单机测试能达到 100w tps
- 低延时: 生产和消费的延时都很低, e2e 的延时在正常的 cluster 中也很低
- 可用性高: replicate+ isr + 选举机制保证
- 工具链成熟: 监控 运维 管理 方案齐全
- 生态成熟: 大数据场景必不可少 kafka stream

### \*不足：\*

- 无法弹性扩容: 对 partition 的读写都在 partition leader 所在的 broker, 如果该 broker 压力过大, 也无法通过新增 broker 来解决问题
- 扩容成本高: 集群中新增的 broker 只会处理新 topic, 如果要分担老 topic-partition 的压力, 需要手动迁移 partition, 这时会占用大量集群带宽
- 消费者新加入和退出会造成整个消费组 rebalance: 导致数据重复消费, 影响消费速度, 增加延迟
- partition 过多会使得性能显著下降: ZK 压力大, broker 上 partition 过多让磁盘顺序写几乎退化成随机写

## 高吞吐机制

### \*顺序存取\*

如果把消息以随机的方式写入到磁盘, 那么磁盘首先要做的就是寻址, 也就是定位到数据所在的物理地址, 在磁盘上就要找到对应柱面、磁头以及对应的扇区; 这个过程相对内存来说会消耗大量时间, 为了规避随机读写带来的时间消耗, kafka 采用顺序写的方式存储数据。

### \*页缓存\*

即使是顺序存取, 但是频繁的 I/O 操作仍然会造成磁盘的性能瓶颈, 所以 kafka 使用了页缓存和零拷贝技术。当进程准备读取磁盘上的文件内容时, 操作系统会先查看待读取的数据是否在页缓存中, 如果存在则直接返回数据, 从而避免了对物理磁盘的 I/O 操作;

如果没有命中, 则操作系统会向磁盘发起读取请求并将读取的数据页存入页缓存, 之后再将数据返回给进程。一个进程需要将数据写入磁盘, 那么操作系统也会检测数据对应的页是否在页缓存中, 如果不存在, 则会先在页缓存中添加相应的页, 最后将数据写入对应的页。被修改过后的页也就变成了脏页, 操作系统会在合适的时间把脏页中的数据写入磁盘, 以保持数据的一致性。

Kafka 中大量使用了页缓存，这是 Kafka 实现高吞吐的重要因素之一。虽然消息都是先被写入页缓存，然后由操作系统负责具体的刷盘任务的，但在 Kafka 中同样提供了同步刷盘及间断性强制刷盘(fsync),可以通过参数来控制。

同步刷盘能够保证消息的可靠性，避免因为宕机导致页缓存数据还未完成同步时造成的数据丢失。但是实际使用上，我们没必要去考虑这样的因素以及这种问题带来的损失，消息可靠性可以由多副本解决，同步刷盘会带来性能的影响。

### 页缓存的好处：

- I/O Scheduler 会将连续的小块写组装成大块的物理写从而提高性能；
- I/O Scheduler 会尝试将一些写操作重新按顺序排好，从而减少磁头移动时间；
- 充分利用所有空闲内存（非 JVM 内存）；
- 读操作可以直接在 Page Cache 内进行，如果消费和生产速度相当，甚至不需要通过物理磁盘交换数据；
- 如果进程重启，JVM 内的 Cache 会失效，但 Page Cache 仍然可用。

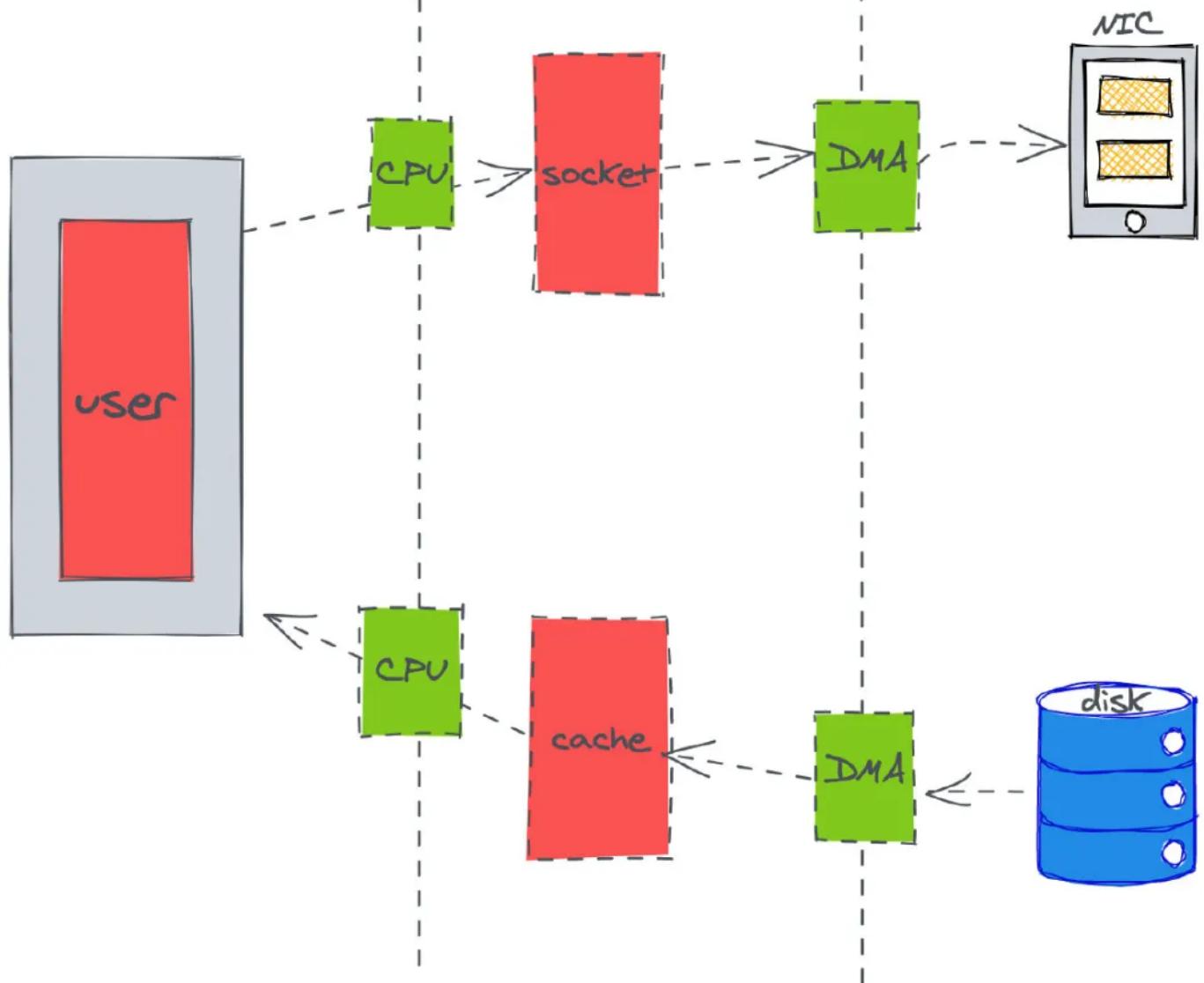
### \*零拷贝\*

零拷贝技术可以减少 CPU 的上下文切换和数据拷贝次数。

### 常规方式

## 用户态

## 内核态



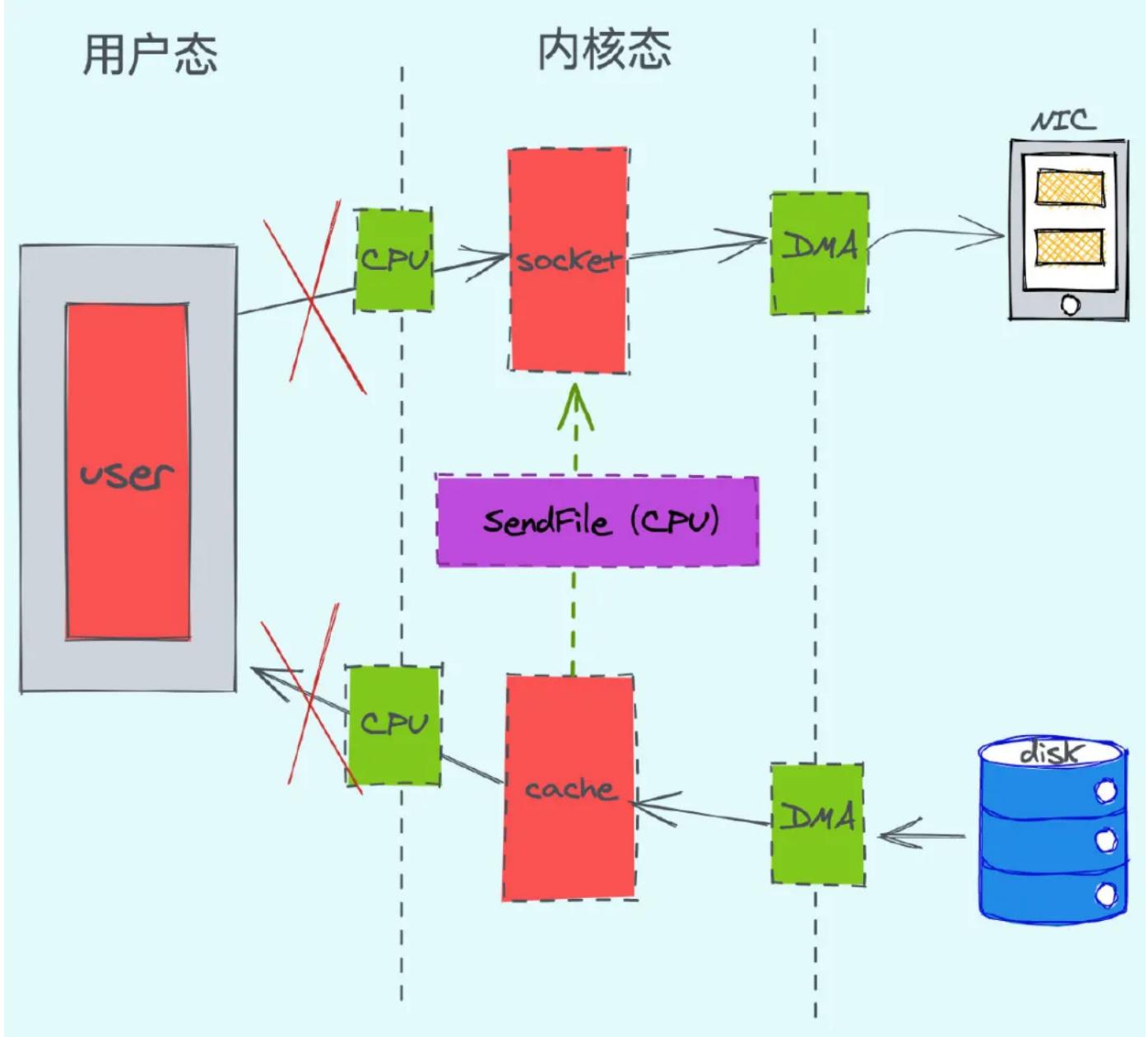
应用程序一次常规的数据请求过程，发生了 4 次拷贝，2 次 DMA 和 2 次 CPU，而 CPU 发生了 4 次的切换。

(DMA 简单理解就是，在进行 I/O 设备和内存的数据传输的时候，数据搬运的工作全部交给 DMA 控制器，而 CPU 不再参与任何与数据搬运相关的事情)

零拷贝的方式

## 用户态

## 内核态



通过零拷贝优化，CPU 只发生了 2 次的上下文切换和 3 次数据拷贝。

### \*批量发送\*

Kafka 允许进行批量发送消息，先将消息缓存在内存中，然后一次请求批量发送出去，这种策略将大大减少服务端的 I/O 次数。

### \*数据压缩\*

Kafka 还支持对消息集合进行压缩，Producer 可以通过 GZIP 或 Snappy 格式对消息集合进行压缩，Producer 压缩之后，在 Consumer 需进行解压，虽然增加了 CPU 的工作，但在对大数据处理上，瓶颈在网络上而不是 CPU，所以这个成本很值得。

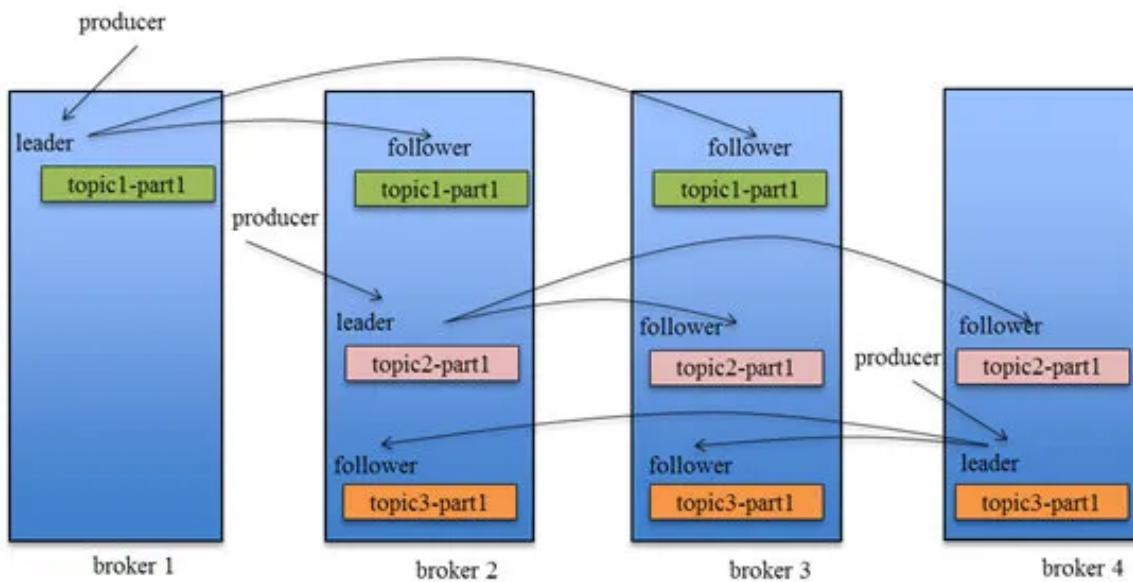
## 高可用机制

### \*副本\*

Producer 在发布消息到某个 Partition 时，先通过 ZooKeeper 找到该 Partition 的 Leader，然后无论该 Topic 的 Replication Factor 为多少，Producer 只将该消息发送到该 Partition 的 Leader。Leader 会将该消息写入其本地 Log。

每个 Follower 都从 Leader pull 数据。这种方式上，Follower 存储的数据顺序与 Leader 保持一致。Follower 在收到该消息后，向 Leader 发送 ACK，并把消息写入其 Log。一旦 Leader 收到了 ISR 中的所有 Replica 的 ACK，该消息就被认为已经 commit 了，Leader 将增加 HW 并且向 Producer 发送 ACK。

为了提高性能，每个 Follower 在接收到数据后就立马向 Leader 发送 ACK，而非等到数据写入 Log 中。因此，对于已经 commit 的消息，Kafka 只能保证它被存于多个 Replica 的内存中，而不能保证它们被持久化到磁盘中，也就不能完全保证异常发生后该条消息一定能被 Consumer 消费。Consumer 读消息也是从 Leader 读取，只有被 commit 过的消息才会暴露给 Consumer。Kafka Replication 的数据流如下图所示：



对于 Kafka 而言，定义一个 Broker 是否“活着”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 的消息复制过来，不能“落后太多”。

Leader 会跟踪与其保持同步的 Replica 列表，该列表称为 ISR（即 in-sync Replica）。如果一个 Follower 宕机，或者落后太多，Leader 将把它从 ISR 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或者 Follower 超过一定时间未向 Leader 发送 fetch 请求。Kafka 的复制机制既不是完全的同步复制，也不是单纯的异步复制。

完全同步复制要求所有能工作的 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）。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 Follower 都复制完都落后于 Leader，而如果 Leader 突然宕机，则会丢失数据。而 Kafka 的这种使用 ISR 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。

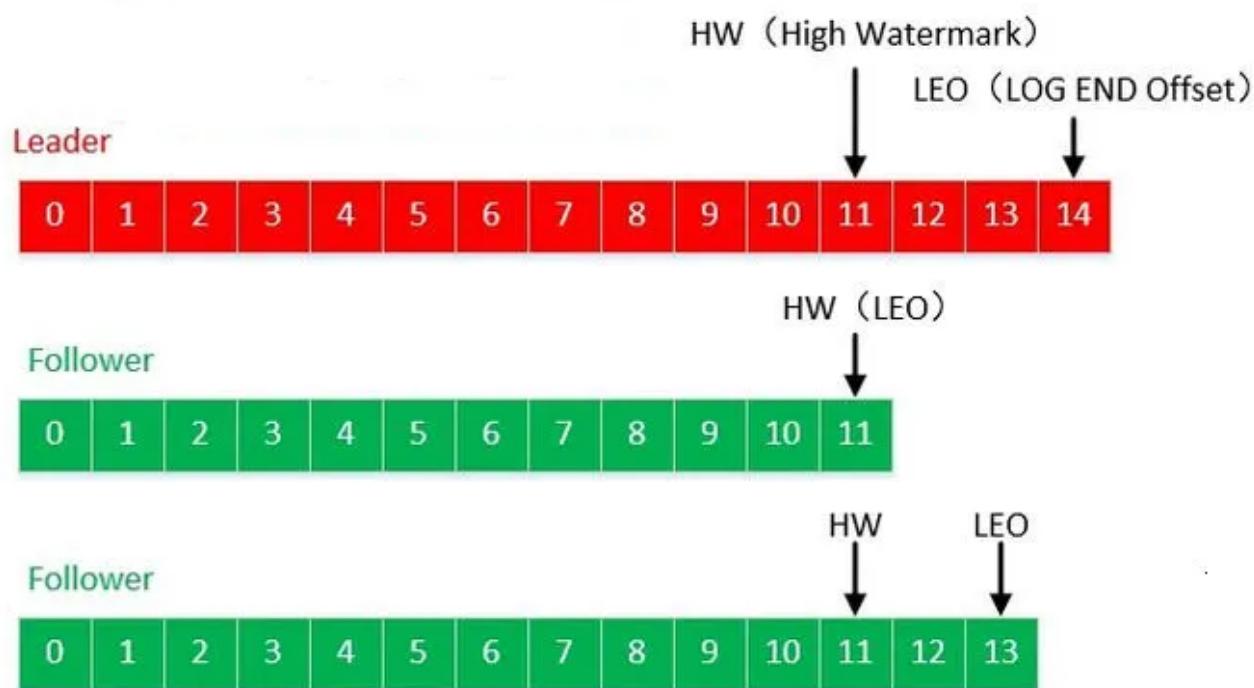
一条消息只有被 ISR 里的所有 Follower 都从 Leader 复制过去才会被认为已提交。这样就避免了部分数据被写进了 Leader，还没来得及被任何 Follower 复制就宕机了，而造成数据丢失（Consumer 无法消费这些数据）。而对于 Producer 而言，它可以选择是否等待消息 commit。这种机制确保了只要 ISR 有一个或以上的 Follower，一条被 commit 的消息就不会丢失。

#### \*故障恢复\*

#### Leader 故障

leader 发生故障后，会从 ISR 中选出一个新的 leader，之后，为保证多个副本之间的数据一致性，其余的 follower 会先将各自的 log 文件高于 HW 的部分截掉，然后从新的 leader 同步数据。注意：这只能保证副本之间的数据一致性，并不能保证数据不丢失或者不重复。

Kafka 在 ZooKeeper 中动态维护了一个 ISR (in-sync replicas)，这个 ISR 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 Partition 能在保证不丢失已经 commit 的消息的前提下容忍  $f$  个 Replica 的失败。



LEO：每个副本最大的 offset。

HW：消费者能见到的最大的 offset，ISR 队列中最小的 LEO。

#### \*Follower\* \*故障\*

follower 发生故障后会被临时踢出 ISR 集合，待该 follower 恢复后，follower 会读取本地磁盘记录的上次的 HW，并将 log 文件高于 HW 的部分截取掉，从 HW 开始向 leader 进行同步数据操作。等该 follower 的 LEO 大于等于该 partition 的 HW，即 follower 追上 leader 后，就可以重新加入 ISR 了。

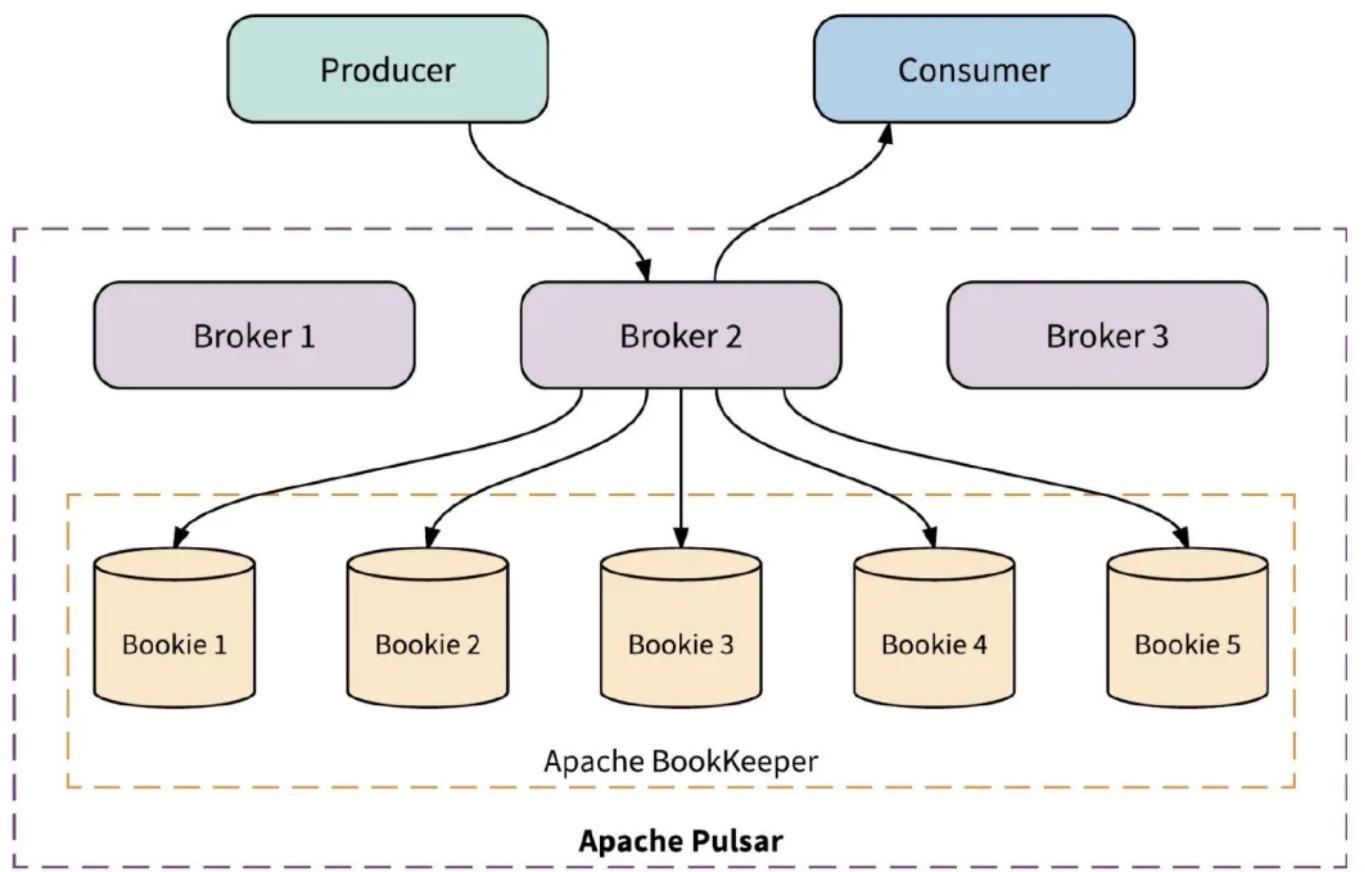
## 扩展性

由于 Broker 存储着特定分区的数据，因此，不管是 Broker 还是分区的扩缩容，都是比较复杂的，属于典型的“有状态服务”扩缩容问题。接下来，我们看一下 Pulsar 是怎么针对 kafka 的不足进行优化的。

## 四、Pulsar

Apache Pulsar 是 Apache 软件基金会顶级项目，是下一代云原生分布式消息流平台，集消息、存储、轻量化函数式计算为一体。采用计算与存储分离架构设计，支持多租户、持久化存储、多机房跨区域数据复制，具有强一致性、高吞吐、低延时及高可扩展性等流数据存储特性。在消息领域，Pulsar 是第一个将存储计算分离云原生架构落地的开源项目。

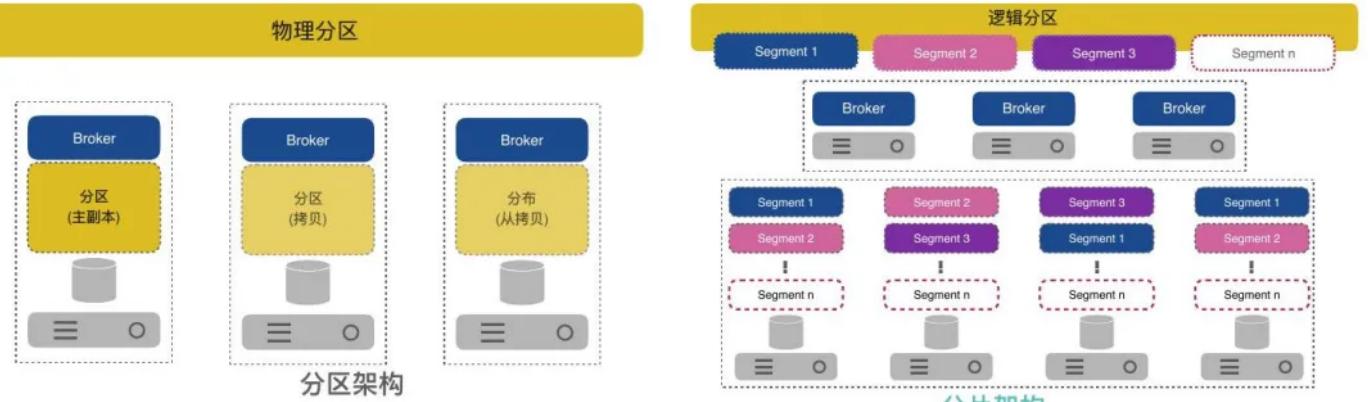
### 服务和存储分离



在 kafka 的基础上，把数据存储功能从 Broker 中分离出来，Broker 仅面向生产者、消费者提供数据读写能力，但其自身并不存储数据。而在 Broker 层下面使用 Bookie 作为存储层，承担具体的数据存储职责。在 Pulsar 中，broker 的含义和 kafka 中的 broker 是一致的，就是一个运行的 Pulsar 实例，提供多个分区的读写服务。由于 broker 层不在承担数据存储职责，使得 Broker 层成为无状态服务。这样一来，Broker 的扩缩容就变得非常简单。

相比之下，服务存储集于一体的 Kafka 就非常难以扩容。

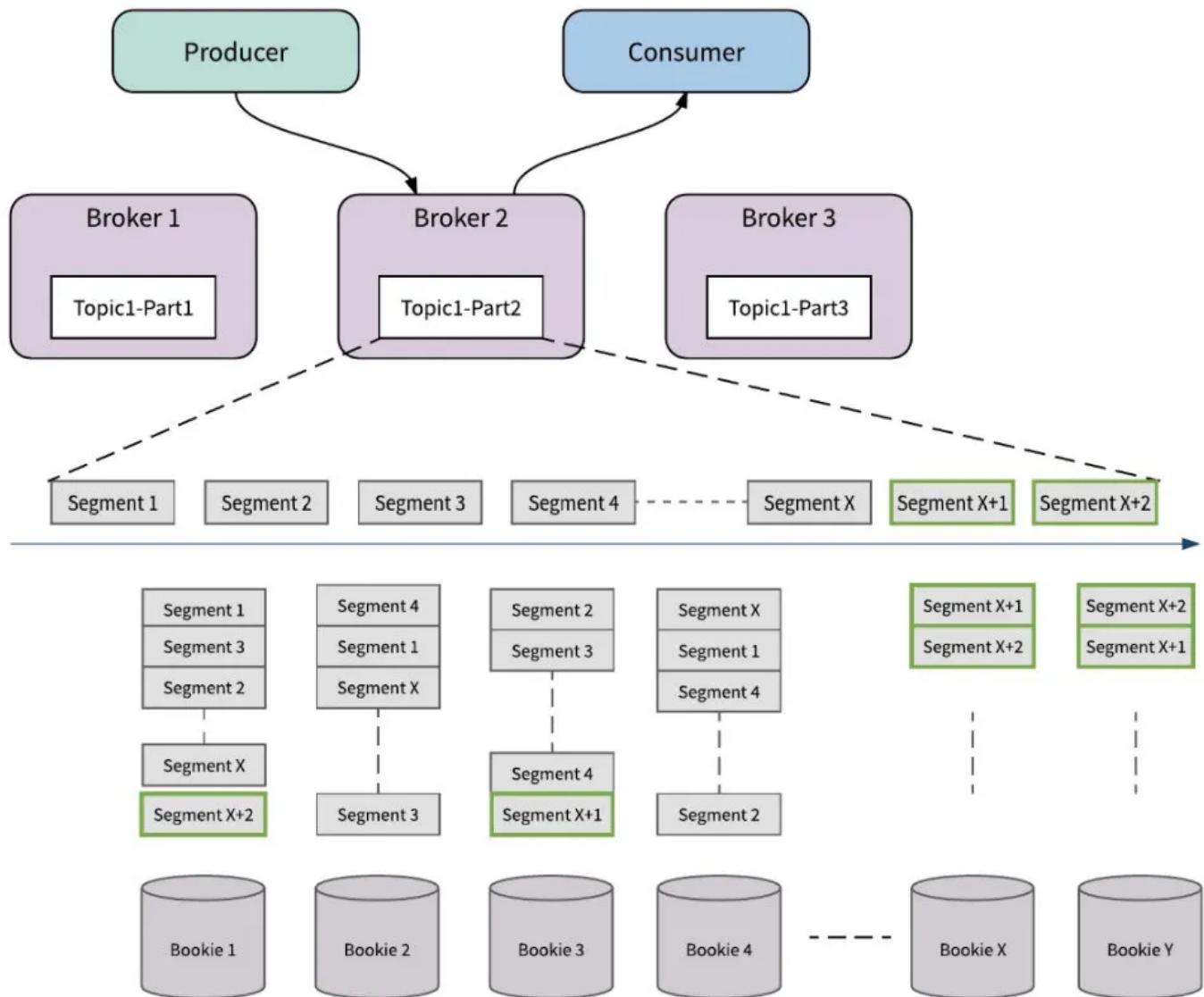
- Broker 和 Bookie 相互独立，方便实现独立的扩展以及独立的容错
- Broker 无状态，便于快速上、下线，更加适合于云原生场景
- 分区存储不受限于单个节点存储容量
- Bookie 数据分布均匀



- 物理分区
- 存储和计算紧耦合
- 容错恢复需要拷贝物理分区
- 扩容需要迁移物理分区来达到负载均衡

- 逻辑分区，“物理”分片
- 存储和计算分离
- 失效处理相互分离，快速、无痛点
- 弹性扩容

## 分片存储



1. 在 Kafka 分区 (Partition) 概念的基础上，按照时间或大小，把分区切分成分片 (Segment)。

2. 同一个分区的分片，分散存储在集群中所有的 Bookie 节点上。

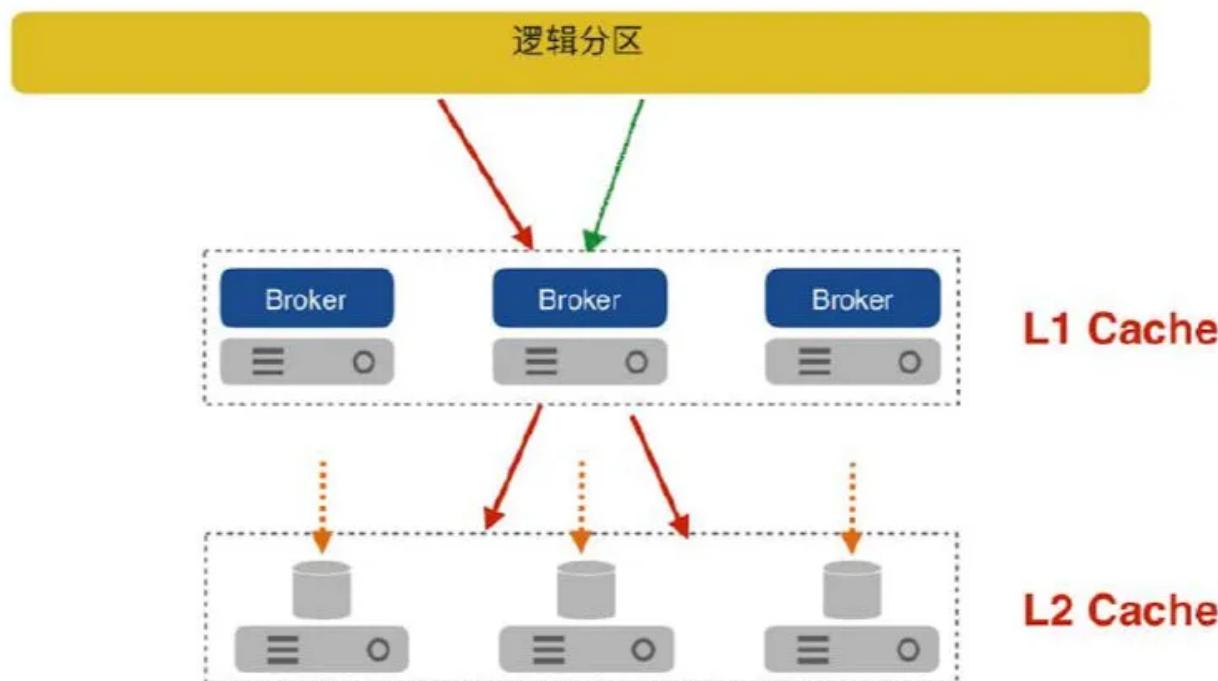
3.同一个分片，拥有多个副本，副本数量可以指定，存储于不同的 Bookie 节点。

## Pulsar 性能

和 Kafka 一样，Pulsar 也使用了顺序读写和零拷贝等技术来提高系统的性能。

此外，Pulsar 还设计了分层缓存机制，在服务层和存储层都做了分层缓存，来提高性能。

- 生产者发送消息时，调用 Bookie 层写入消息时，同时将消息写入 broker 缓存中。
- 实时消费时（追尾读），首先从 broker 缓存中读取数据，避免从持久层 bookie 中读取，从而降低投递延迟。
- 读取历史消息（追赶读）场景中，bookie 会将磁盘消息读入 bookie 读缓存中，从而避免每次都读取磁盘数据，降低读取延时。



## Pulsar 扩展性

分片存储解决了分区容量受单节点存储空间限制的问题，当容量不够时，可以通过扩容 Bookie 节点的方式支撑更多的分区数据，也解决了分区数据倾斜问题，数据可以均匀的分配在 Bookie 节点上。

Broker 和 Bookie 灵活的容错以及无缝的扩容能力让 Apache Pulsar 具备非常高的可用性，实现了无限制的分区存储。

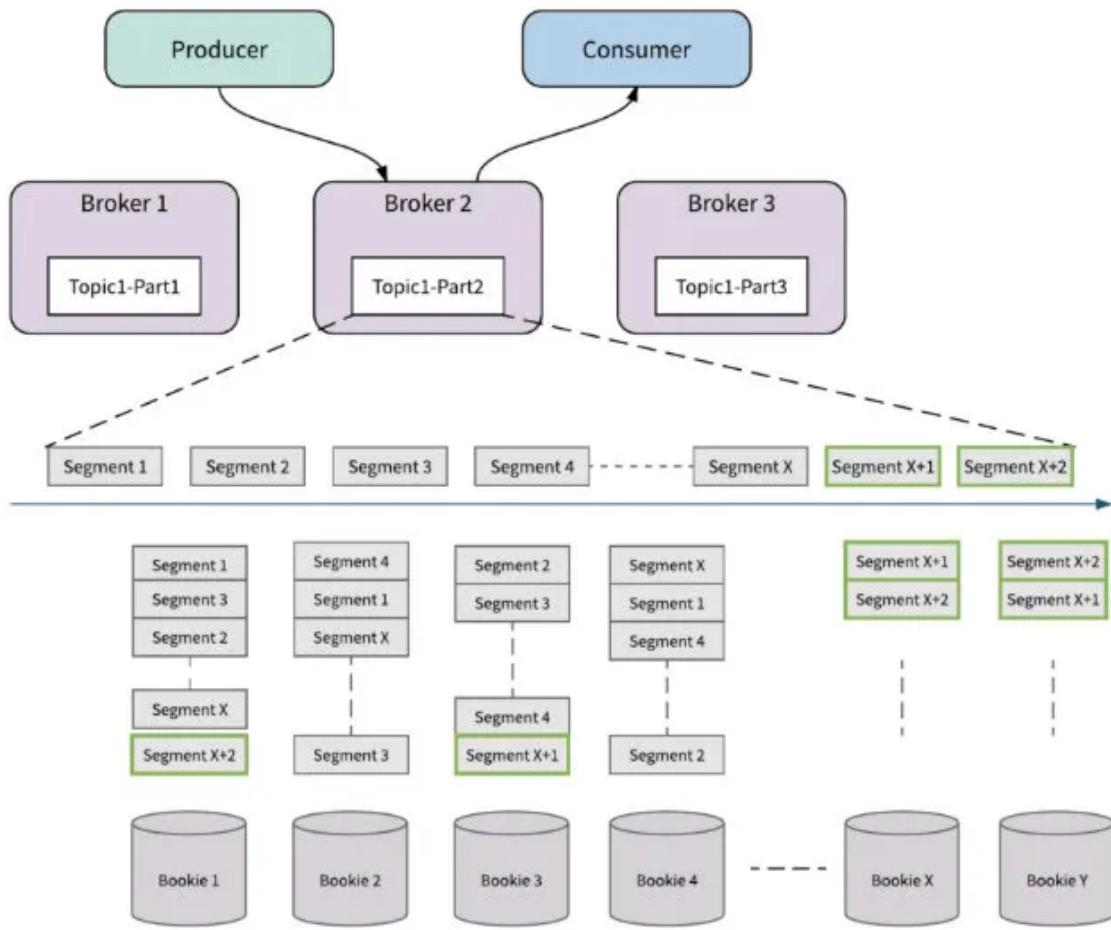


### \*Broker\* \*扩展\*

在 Pulsar 中 Broker 是无状态的，可以通过增加节点的方式实现快速扩容。当需要支持更多的消费者或生产者时，可以简单地添加更多的 Broker 节点来满足业务需求。Pulsar 支持自动的分区负载均衡，在 Broker 节点的资源使用率达到阈值时，会将负载迁移到负载较低的 Broker 节点。新增 Broker 节点时，分区也将在 Brokers 中做平衡迁移，一些分区的所有权会转移到新的 Broker 节点。

### Bookie 扩展

存储层的扩容，通过增加 Bookie 节点来实现。通过资源感知和数据放置策略，流量将自动切换到新的 Apache Bookie 中，整个过程不会涉及到不必要的数据搬迁。即扩容时，不会将旧数据从现有存储节点重新复制到新存储节点。



如图所示，起始状态有四个存储节点，Bookie1, Bookie2, Bookie3, Bookie4，以 Topic1-Part2 为例，当这个分区的最新的存储分片是 SegmentX 时，对存储层扩容，添加了新的 Bookie 节点，BookieX, BookieY，那么当存储分片滚动之后，新生成的存储分片，SegmentX+1, SegmentX+2，会优先选择新的 Bookie 节点（BookieX, BookieY）来保存数据。

## Pulsar 可用性

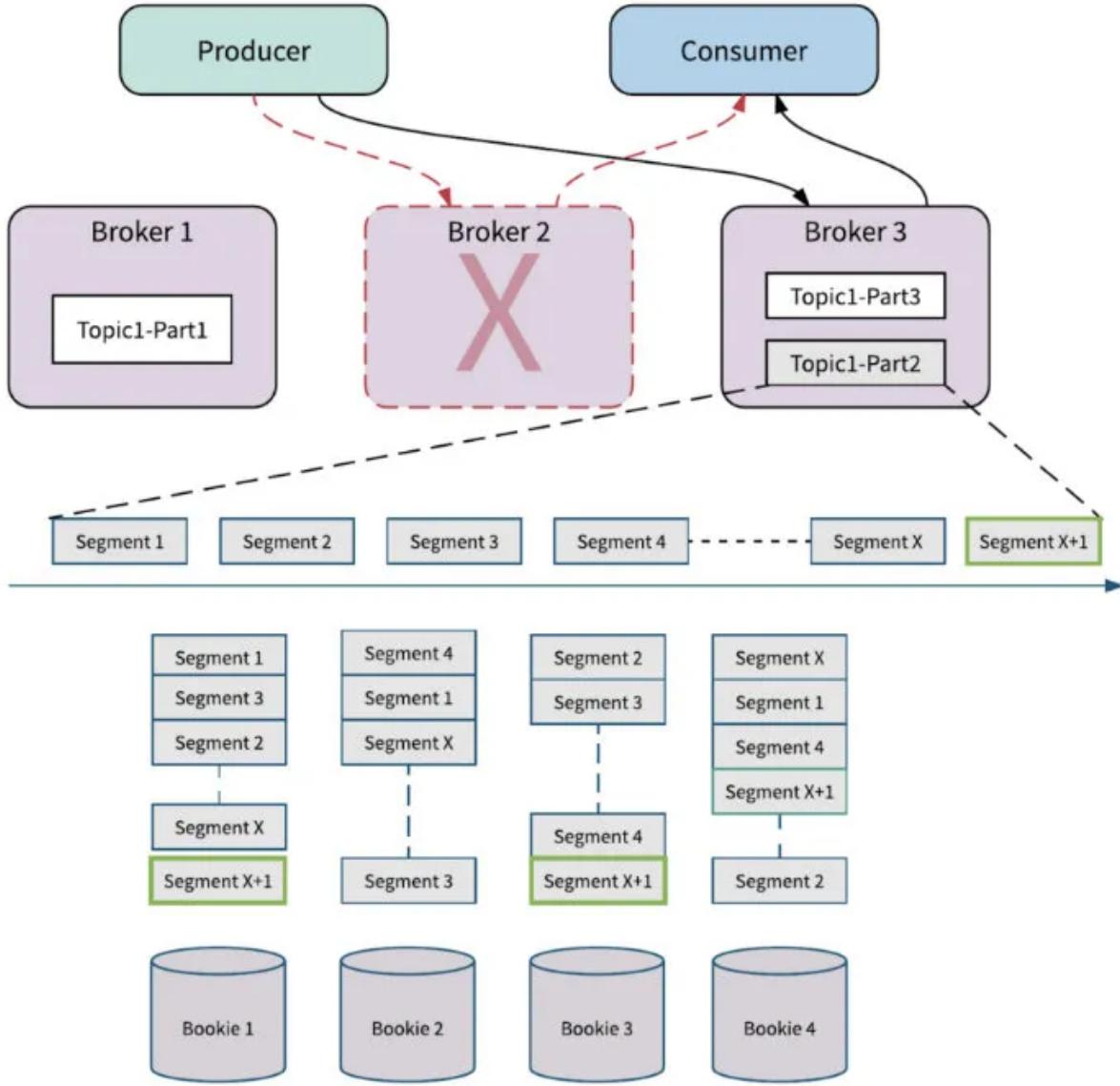
### \*Broker\* \*容错\*

如下图，假设当存储分片滚动到 SegmentX 时，Broker2 节点失败。此时生产者和消费者向其他的 Broker 发起请求，这个过程会触发分区的所有权转移，即将 Broker2 拥有的分区 Topic1-Part2 的所有权转移到其他的 Broker(Broker3)。

由于数据存储和数据服务分离，所以新 Broker 接管分区的所有权时，它不需要复制 Partition 的数据。新的分区 Owner (Broker3) 会产生一个新的分片 SegmentX+1，如果有新数据到来，会存储在新的分片 Segment X+1 上，不会影响分区的可用性。

即当某个 Broker 实例故障时，整个集群的消息存储能力仍然完好。此时，集群只是丧失了特定分区的消息服务，只需要把这些分区的服务权限分配给其他 Broker 即可。

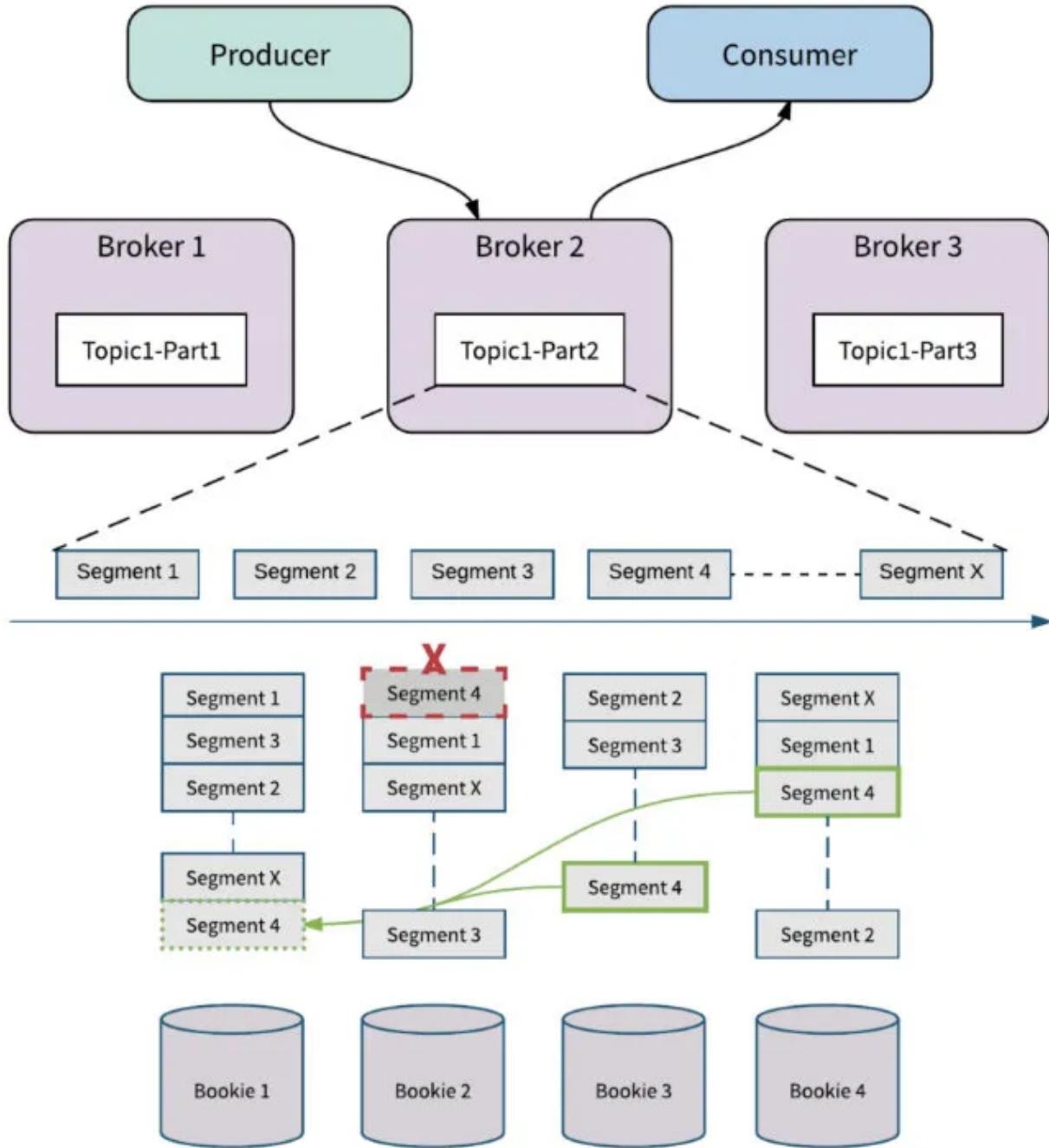
注意，和 Kafka 一样，Pulsar 的一个分区仍然只能由一个 Broker 提供服务，否则无法保证消息的分区有序性。



## Bookie 容错

如下图，假设 Bookie 2 上的 Segment 4 损坏。Bookie Auditor 会检测到这个错误并进行修复。Bookie 中的副本修复是 Segment 级别的多对多快速修复，BookKeeper 可以从 Bookie 3 和 Bookie 4 读取 Segment 4 中的消息，并在 Bookie 1 处修复 Segment 4。如果是 Bookie 节点故障，这个 Bookie 节点上所有的 Segment 会按照上述方式复制到其他的 Bookie 节点。

所有的副本修复都在后台进行，对 Broker 和应用透明，**Broker 会产生新的 Segment 来处理写入请求，不会影响分区的可用性。**



## Pulsar 其他特性

基于上述的设计特点，Pulsar 提供了很多特性。

### \*读写分离\*

Pulsar 另外一个有吸引力的特性是提供了读写分离的能力，读写分离保证了在有大量滞后消费（磁盘 IO 会增加）时，不会影响服务的正常运行，尤其是不会影响到数据的写入。读写分离的能力由 Bookie 提供，简单说一下 Bookie 存储涉及到的概念：

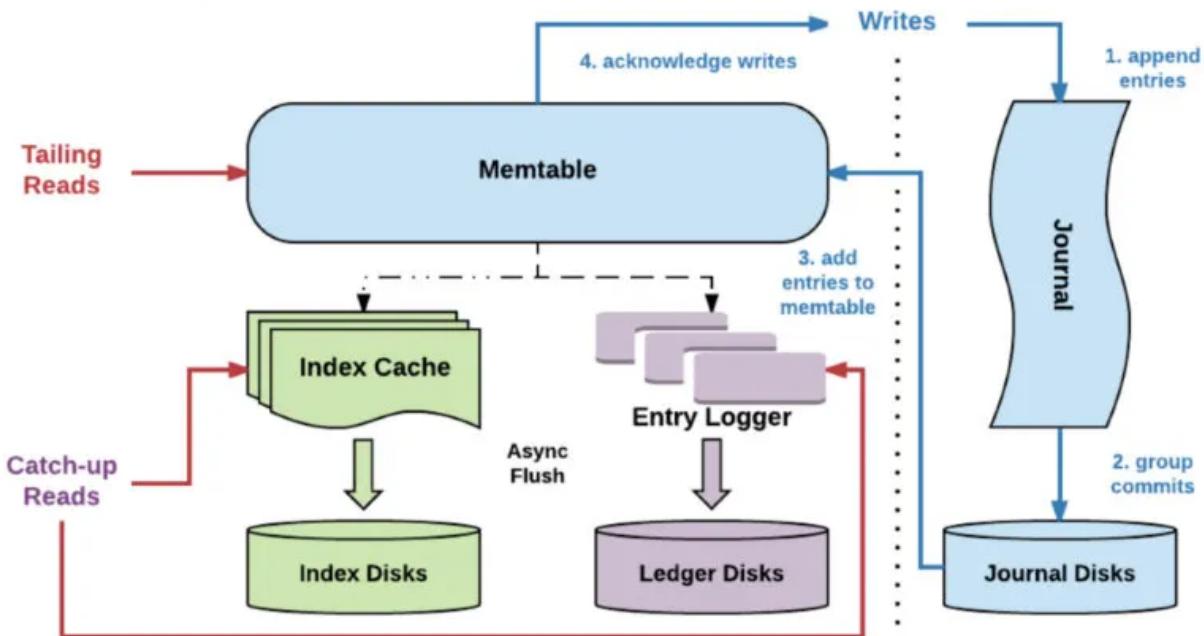
- Journals: Journal 文件包含了 Bookie 事务日志，在 Ledger (可以认为是分片的一部分) 更新之前，Journal 保证描述更新的事务写入到 Non-volatile 的存储介质上；
- Entry logger: Entry 日志文件管理写入的 Entry，来自不同 ledger 的 entry 会被聚合然后顺序写入；
- Index files: 每个 Ledger 都有一个对应的索引文件，记录数据在 Entry 日志文件中的 Offset 信息。

Entry 的读写入过程下图所示，数据的写入流程：

- 数据首先会写入 Journal，写入 Journal 的数据会实时落到磁盘；
- 然后，数据写入到 Memtable，Memtable 是读写缓存；
- 写入 Memtable 之后，对写入请求进行响应；
- Memtable 写满之后，会 Flush 到 Entry Logger 和 Index cache，Entry Logger 中保存了数据，Index cache 保存了数据的索引信息，然后由后台线程将 Entry Logger 和 Index cache 数据落到磁盘。

数据的读取流程：

- 如果是 Tailing read 请求，直接从 Memtable 中读取 Entry；
- 如果是 Catch-up read（滞后消费）请求，先读取 Index 信息，然后索引从 Entry Logger 文件读取 Entry。



一般在进行 Bookie 的配置时，会将 Journal 和 Ledger 存储磁盘进行隔离，减少 Ledger 对于 Journal 写入的影响，并且推荐 Journal 使用性能较好的 SSD 磁盘，读写分离主要体现在：

- 写入 Entry 时，Journal 中的数据需要实时写到磁盘，Ledger 的数据不需要实时落盘，通过后台线程批量落盘，因此写入的性能主要受到 Journal 磁盘的影响；
- 读取 Entry 时，首先从 Memtable 读取，命中则返回；如果不命中，再从 Ledger 磁盘中读取，所以对于 Catch-up read 的场景，读取数据会影响 Ledger 磁盘的 IO，对 Journal 磁盘没有影响，也就不会影响到数据的写入。

所以，数据写入主要是受 Journal 磁盘的负载影响，不会受 Ledger 磁盘的影响。另外，Segment 存储的多个副本都可以提供读取服务，相比于主从副本的设计，Apache Pulsar 可以提供更好的数据读取能力。

通过以上分析，Apache Pulsar 使用 Apache BookKeeper 作为数据存储，可以带来下列的收益：

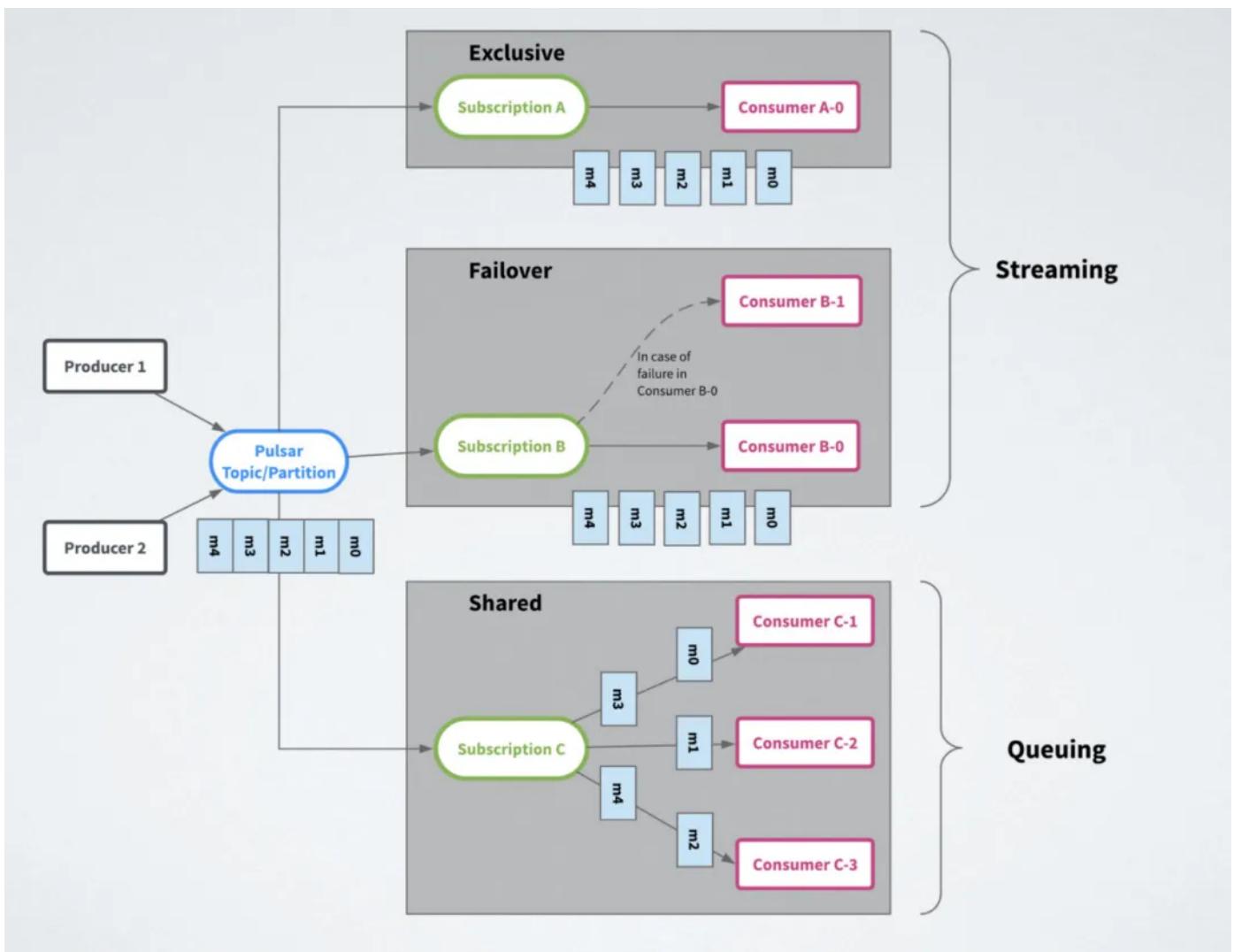
- 支持将多个 Ledger 的数据写入到同一个 Entry logger 文件，可以避免分区膨胀带来的性能下降问题
- 支持读写分离，可以在滞后消费场景导致磁盘 IO 上升时，保证数据写入的不受影响
- 支持全副本读取，可以充分利用存储副本的数据读取能力

## \*多种消费模型\*

Pulsar 提供了多种订阅方式来消费消息，分为三种类型：独占（Exclusive），故障切换（Failover）或共享（Share）。

- **Exclusive 独占订阅**：在任何时间，一个消费者组（订阅）中有且只有一个消费者来消费 Topic 中的消息。
- **Failover 故障切换**：多个消费者（Consumer）可以附加到同一订阅。但是，一个订阅中的所有消费者，只会有一个消费者被选为该订阅的主消费者。其他消费者将被指定为故障转移消费者。当主消费者断开连接时，分区将被重新分配给其中一个故障转移消费者，而新分配的消费者将成为新的主消费者。发生这种情况时，所有未确认（ack）的消息都将传递给新的主消费者。
- **Share 共享订阅**：使用共享订阅，在同一个订阅背后，用户按照应用的需求挂载任意多的消费者。订阅中的所有消息以循环分发形式发送给订阅背后的多个消费者，并且一个消息仅传递给一个消费者。

当消费者断开连接时，所有传递给它但是未被确认（ack）的消息将被重新分配和组织，以便发送给该订阅上剩余的剩余消费者。

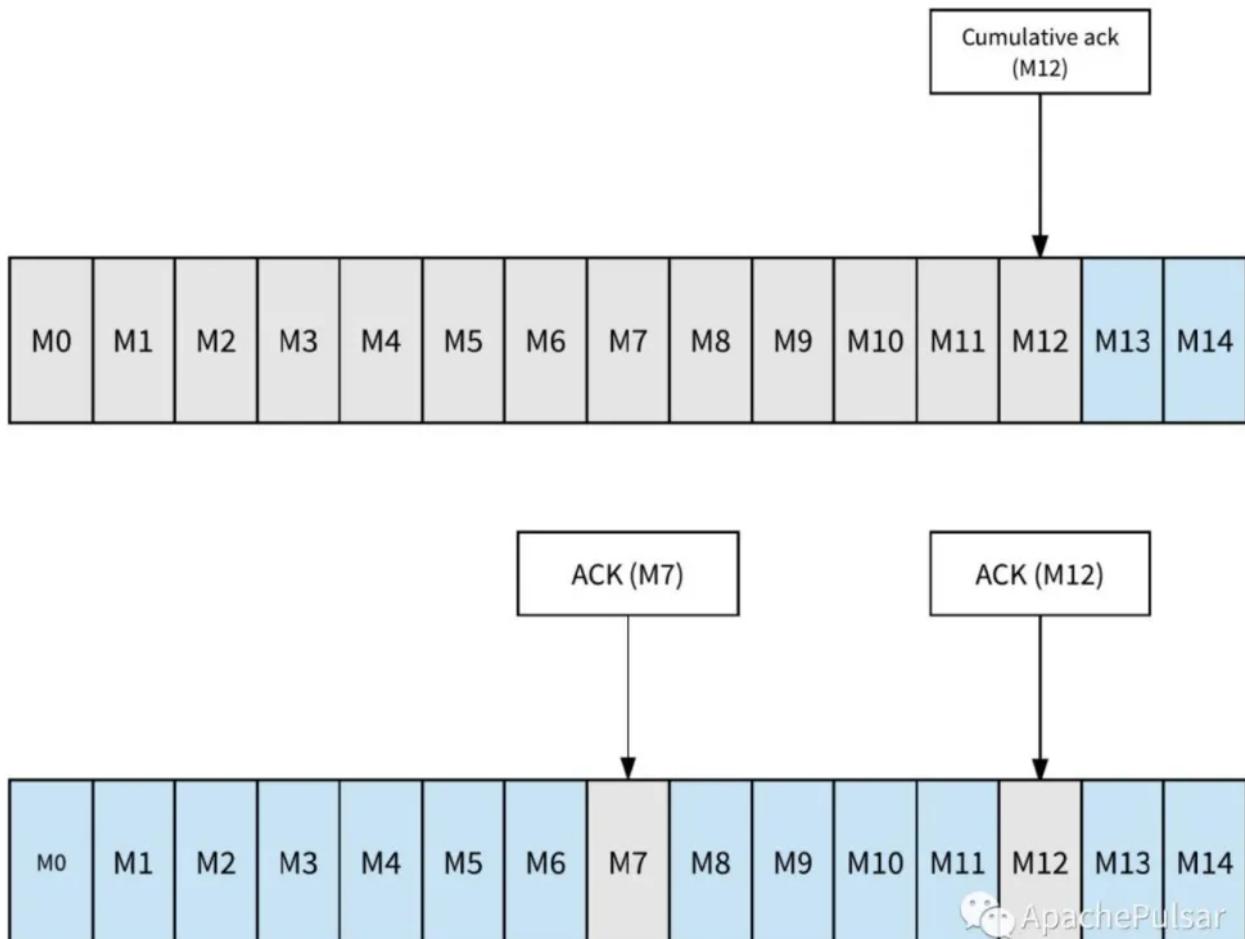


## \*多种 ACK 模型\*

消息确认（ACK）的目的就是保证当发生故障后，消费者能够从上一次停止的地方恢复消费，保证既不会丢失消息，也不会重复处理已经确认（ACK）的消息。在 Pulsar 中，每个订阅中都使用一个专门的数据结构-游标（Cursor）来跟踪订阅中的每条消息的确认（ACK）状态。每当消费者在分区上确认消息时，游标都会更新。

Pulsar 提供两种消息确认方法：

- 单条确认（Individual Ack），单独确认一条消息。被确认后的消息将不会被重新传递
- 累积确认（Cumulative Ack），通过累积确认，消费者只需要确认它收到的最后一条消息



上图说明了单条确认和累积确认的差异（灰色框中的消息被确认并且不会被重新传递）。对于累计确认，M12 之前的消息被标记为 Acked。对于单独进行 ACK，仅确认消息 M7 和 M12，在消费者失败的情况下，除了 M7 和 M12 之外，其他所有消息将被重新传送。