深入理解 Kafka: 核心设计与实现原理

王社英

北京回龙观

April 16, 2020

Outline

Outline

Kafka 简介

Kafka 定位为一个分布式流式处理平台, 具有以下特性:

- 高吞吐
- 可持久化
- 可水平扩展
- 支持流数据处理

Kafka 扮演的三大角色

1. 消息系统
2. 存储系统
3. 流式处理平台

基本概念

体系架构

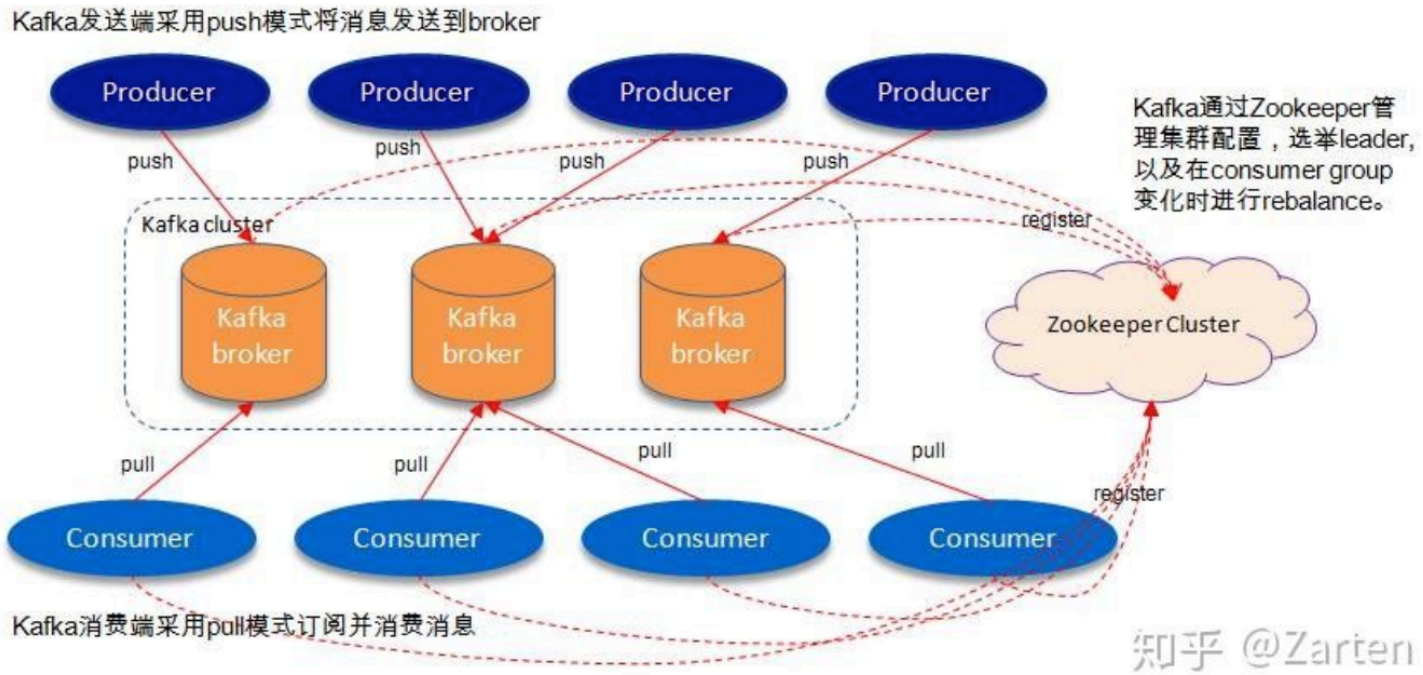


Figure 1: kafka 体系架构图

基本概念

体系架构

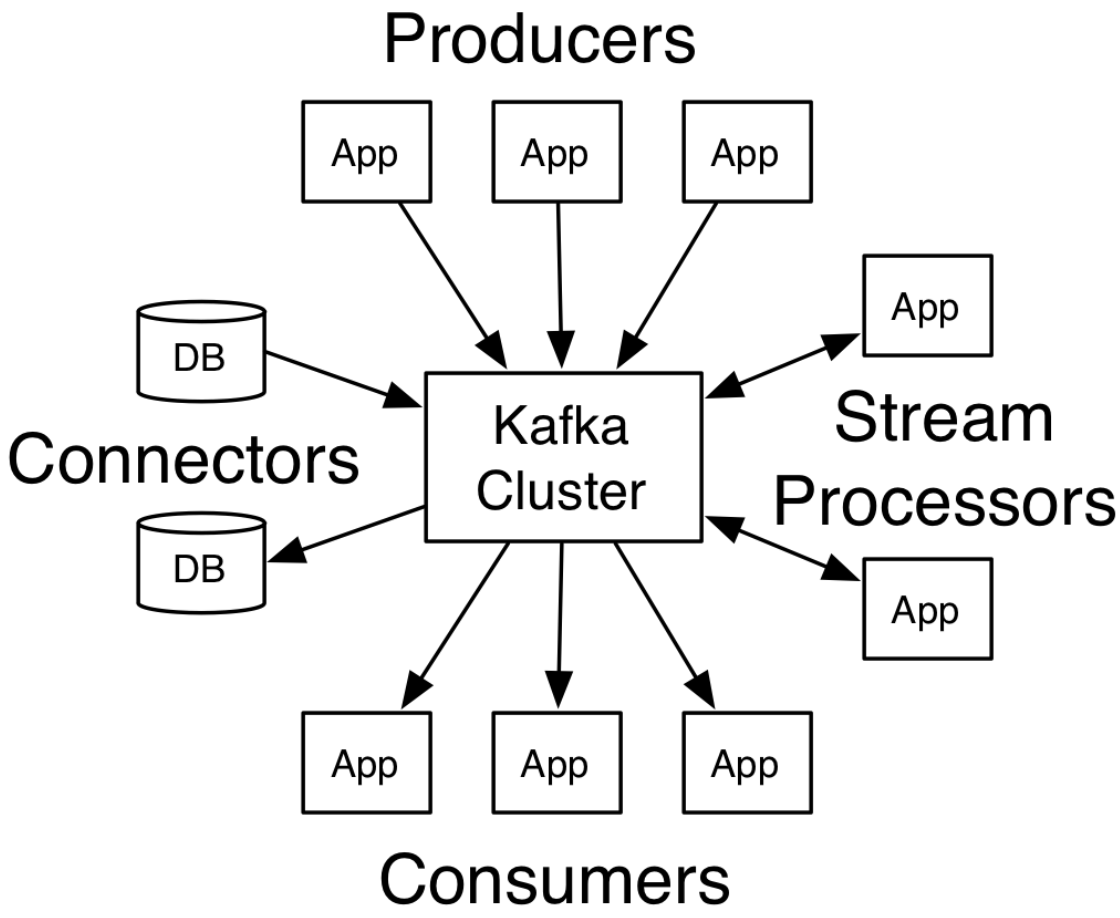


Figure 2: kafka 体系架构图

基本概念

体系架构

整个 Kafka 体系结构分 3 部分:

- Producer
- Consumer
- Broker

Kafka 的三层消息架构:

- Topic
- Partition
- Record

基本概念

体系架构

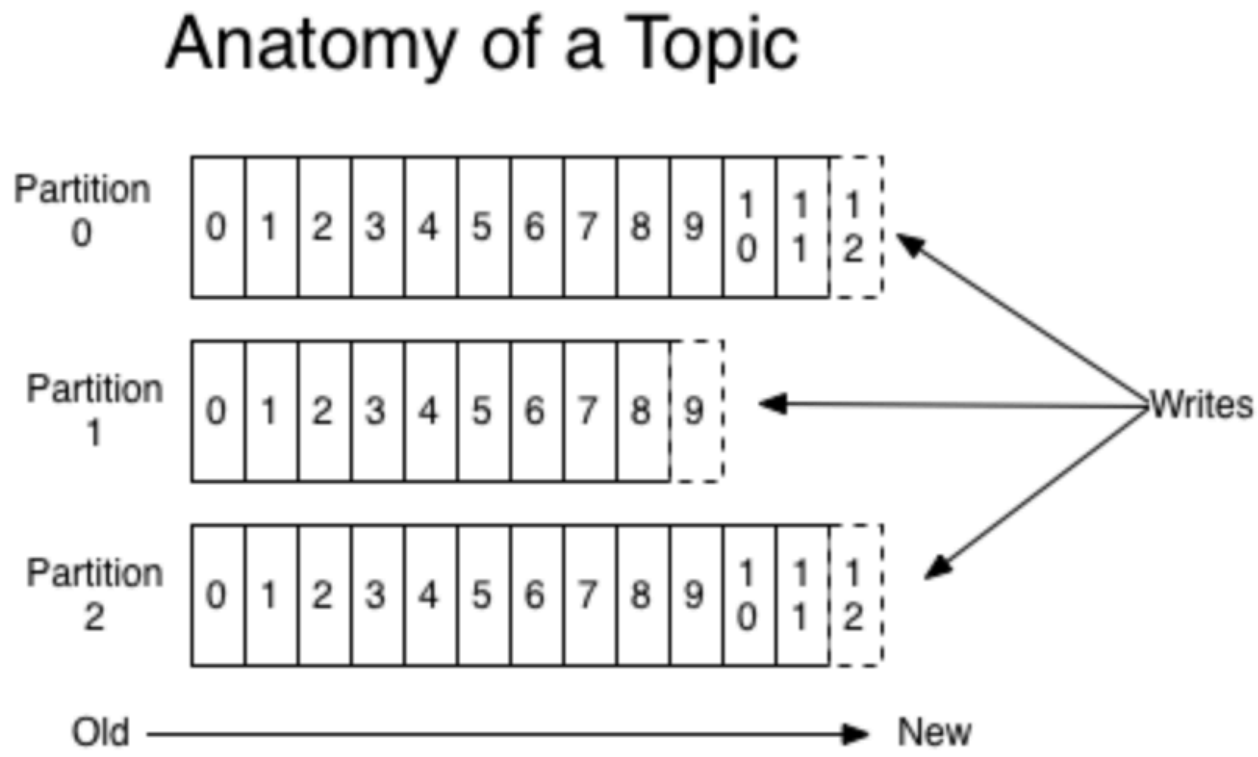


Figure 3: 消息追加--分区视角

基本概念

体系架构

Figure 4: 消息追加--生产者视角

基本概念

高可用

Kafka 高可用机制:

- AR(Assigned Replicas)
- ISR(In-Sync Replicas)
- OSR(Out-of-Sync Replicas)

在正常情况下, 所有的 follower 副本都应该与 leader 副本保持一定程度的同步 AR=ISR,OSR=∅

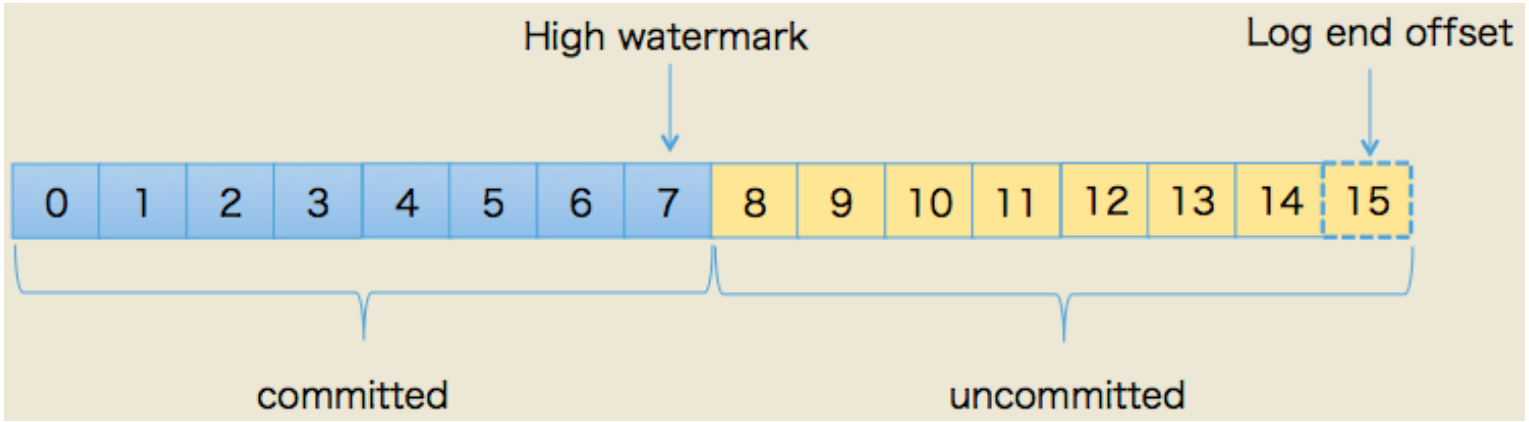


Figure 5: HW 和 LEO

基本概念

数据一致性

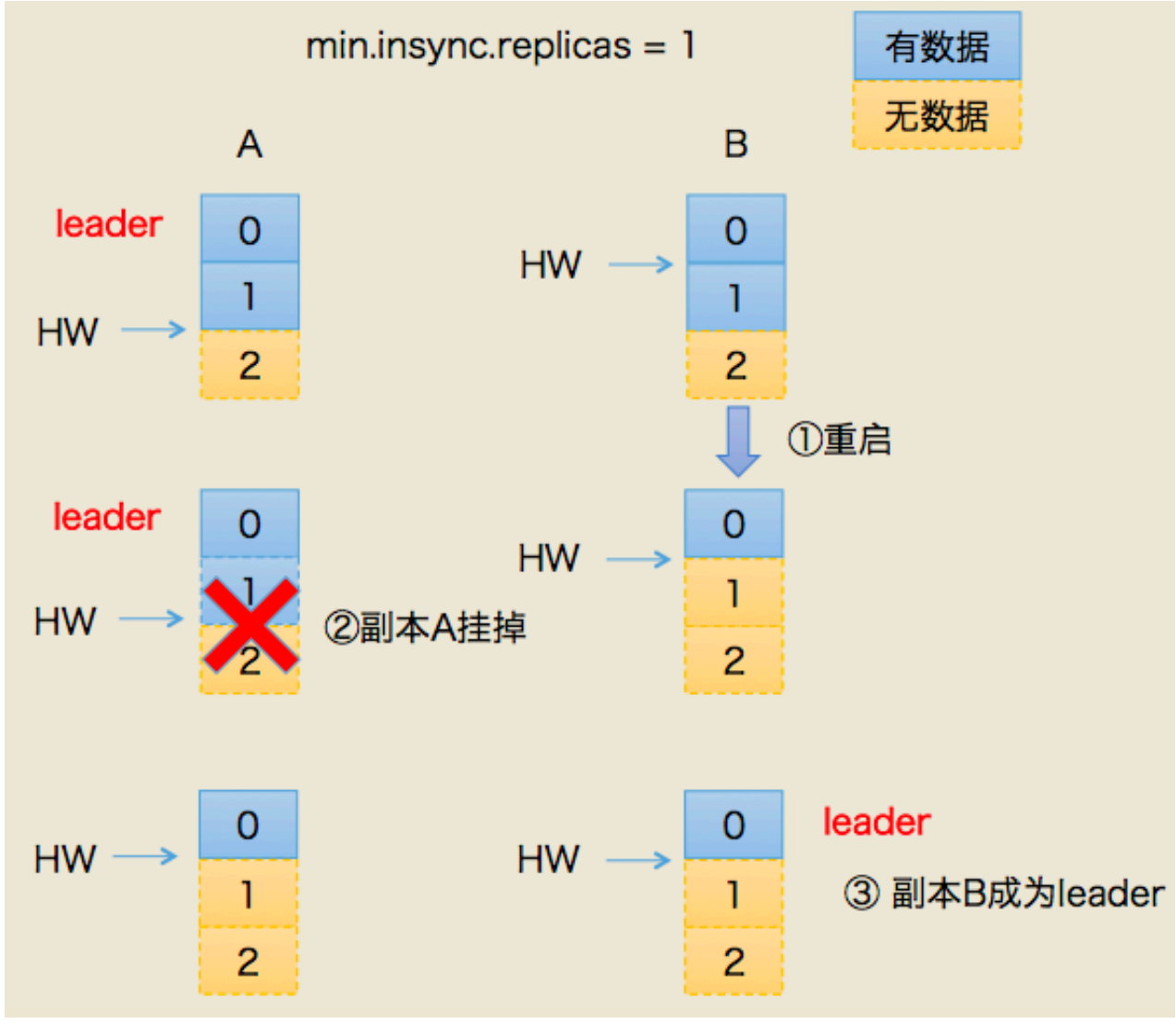


Figure 6: 数据丢失

基本概念

数据一致性

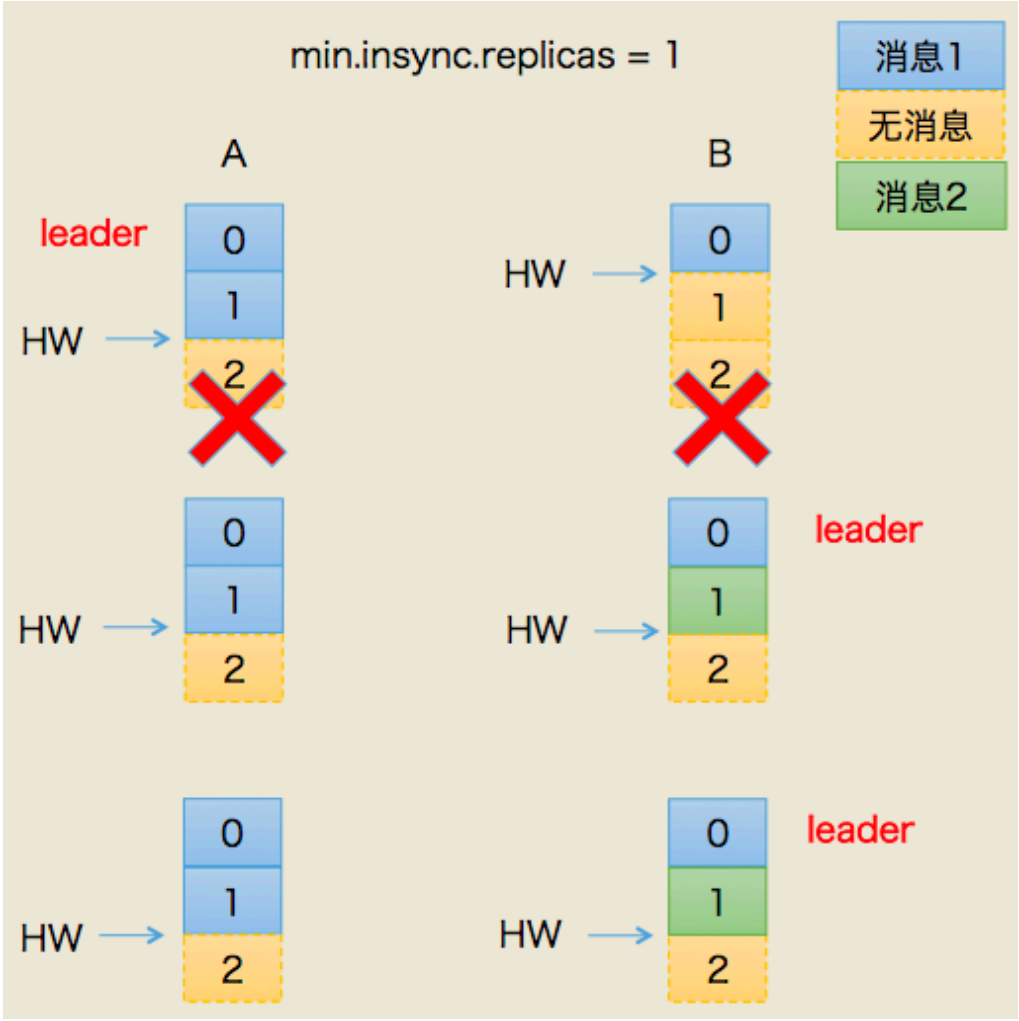


Figure 7: 数据不一致

基本概念

数据一致性

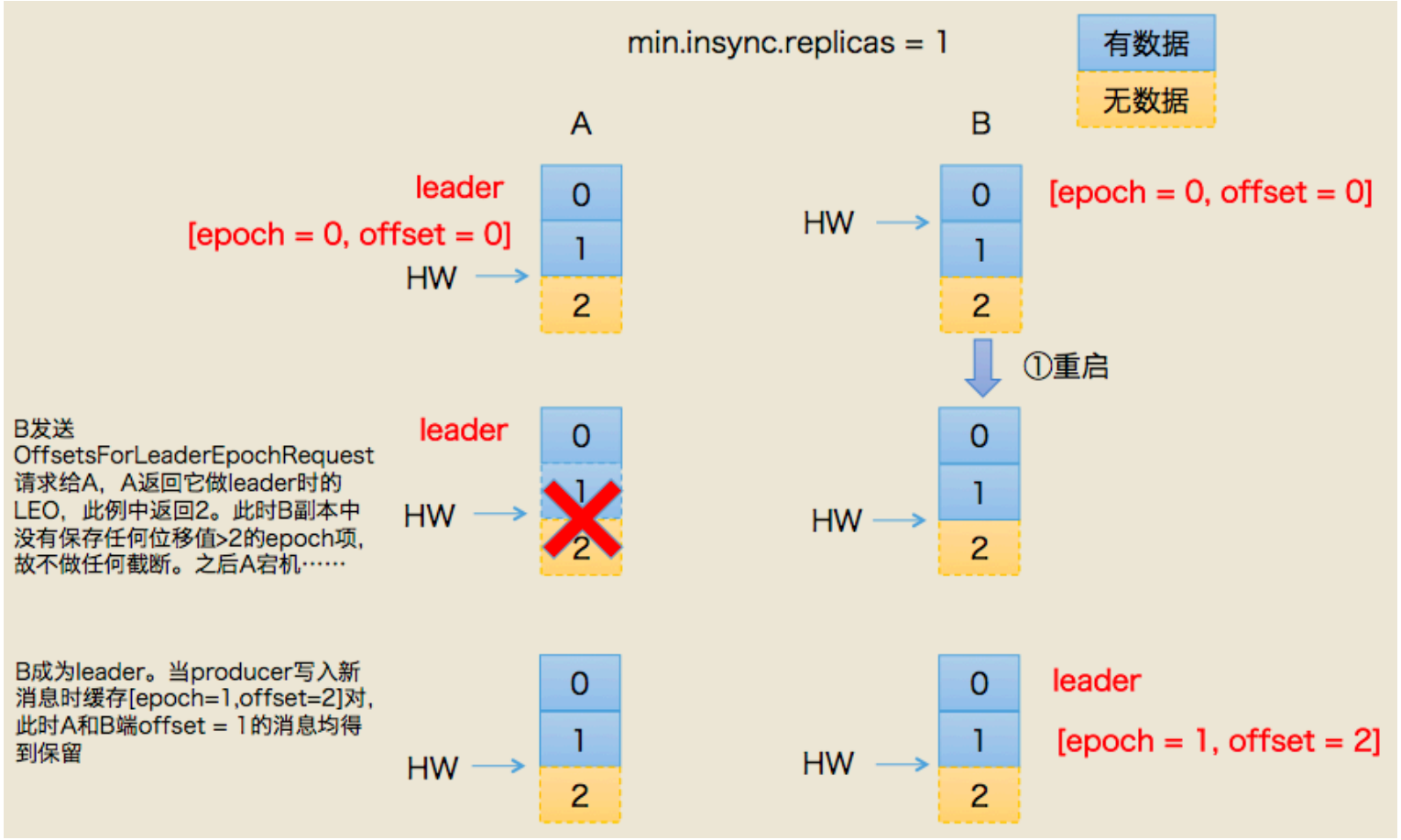


Figure 8: leader epoch 规避数据丢失

基本概念

数据一致性

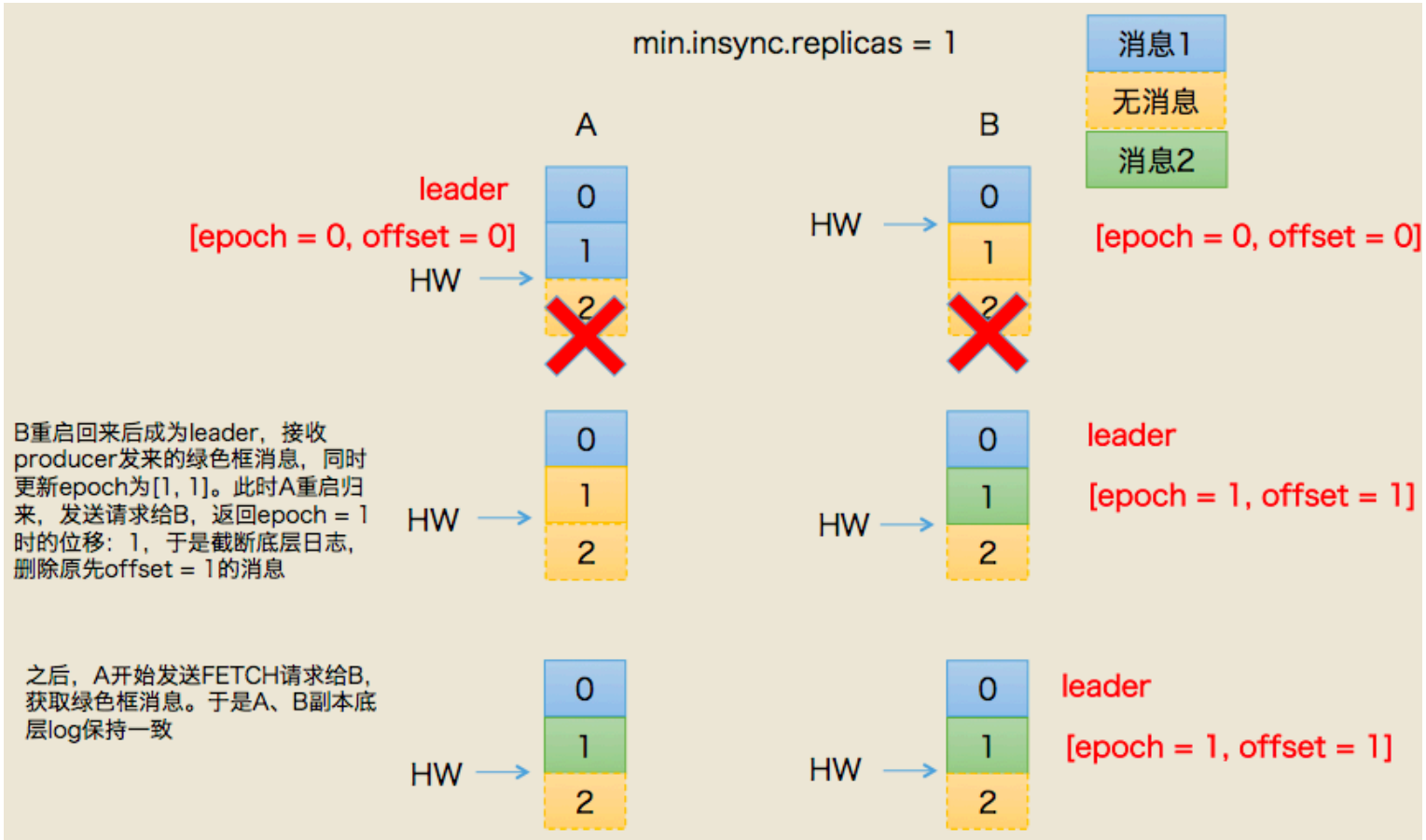


Figure 9: leader epoch 规避数据不一致

Outline

原理分析

消息的发送

KafkaProducer 是线程安全的, 可以在多个线程中共享单个 KafkaProducer 实例, 也可以将 KafkaProducer 实例进行池化来供其他线程调用.

生产者发送消息的三种模式:

- fire-and-forget
- sync
- async

  KafkaProducer.send() 返回值是 Future<RecordMetadata>. Future 表示一个任务的生命周期, 提供了相应的方法判断
- 任务状态, 完成或取消
- 任务结果
- 取消任务

原理分析

消息的发送

生产者发送消息常见异常:

- 可重试异常
  - NetworkException,
  - LeaderNotAvailableException
  - UnknownTopicOrPartitionException
  - NotEnoughReplicasException
  - NotCoordinatorException
- 不可重试异常
  - RecordTooLargeException

原理分析

消息的发送

生产者需要用序列化器 (Serializer) 把对象转换成字节数组才能通过网络发送给 Kafka.

消息在通过 send() 方法发往 broker 的过程中, 有可能需要经过拦截器 (Interceptor,可选), 序列化器 (Serializer,必选) 和分区器 (Partitioner,默认轮询) 的一系列作用之后才能被真正发往 broker.

原理分析

整体架构

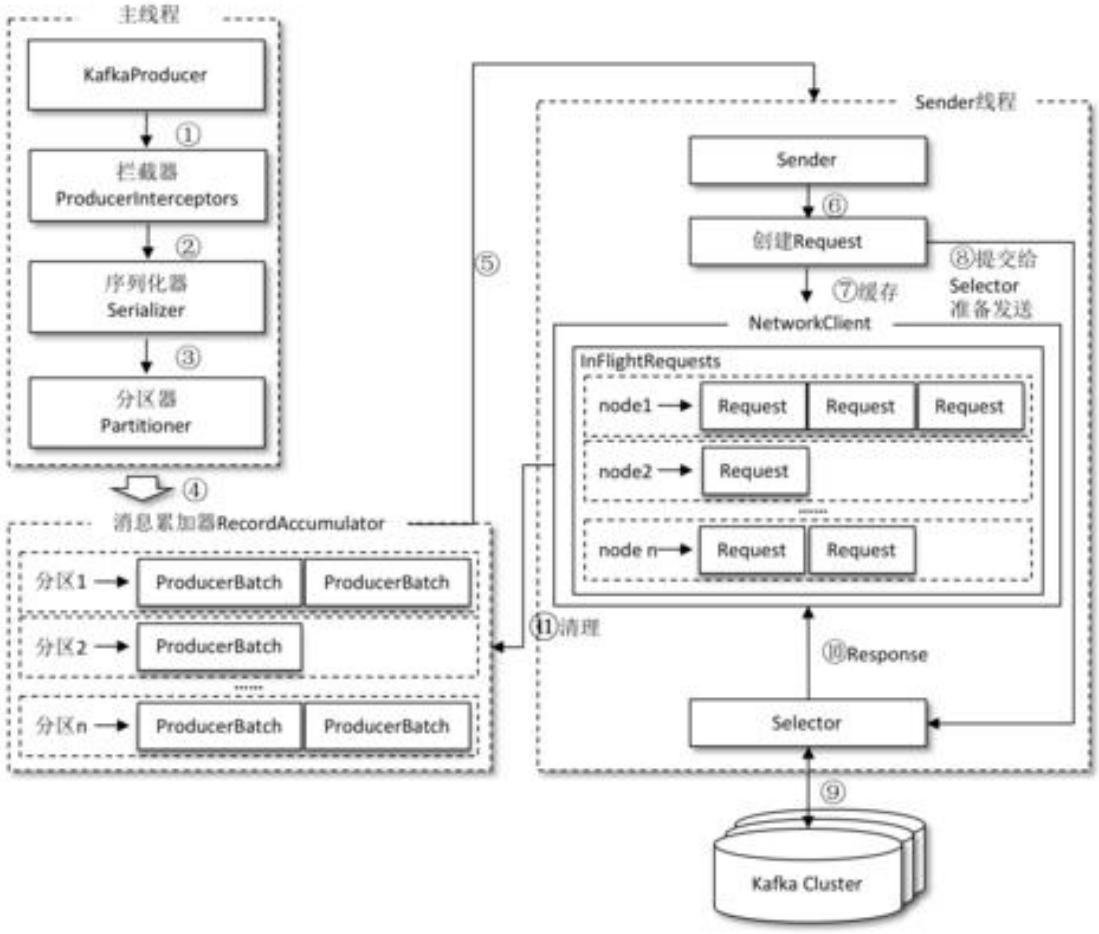
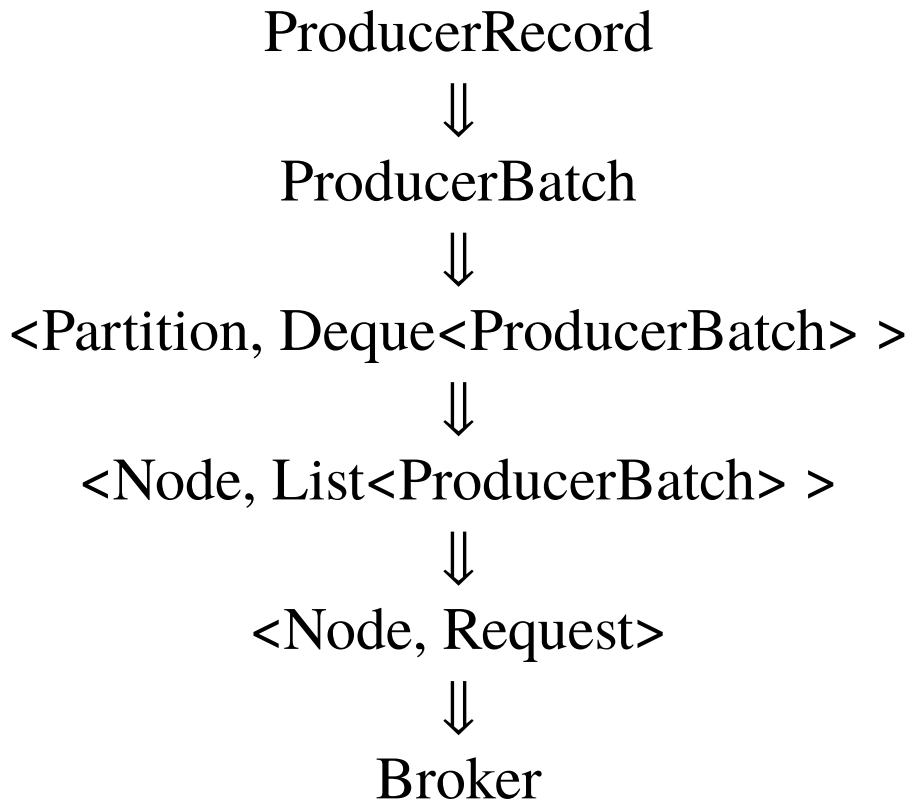


Figure 10: 生产者客户端整体架构

原理分析

整体架构


ProducerRecord
⇓
ProducerBatch
⇓
<Partition, Deque<ProducerBatch> >
⇓
<Node, List<ProducerBatch> >
⇓
<Node, Request>
⇓
Broker

原理分析

整体架构

元数据是指 Kafka 集群的元数据, 这些元数据具体记录了集群中有哪些 Topic, 这些 Topic 有哪些 Partition, 每个 Partition 的 leader 副本分配在哪个 Broker 上,follower 副本分配在哪些 Broker 上, 哪些副本在 AR,ISR 集合中, 集群有哪些 Broker, 控制器 Broker 又是哪一个等信息.

当客户端中没有这些元数据信息时, 或者超过 metadata.max.age.ms(默认 5 分钟) 时, 都会更新元数据.

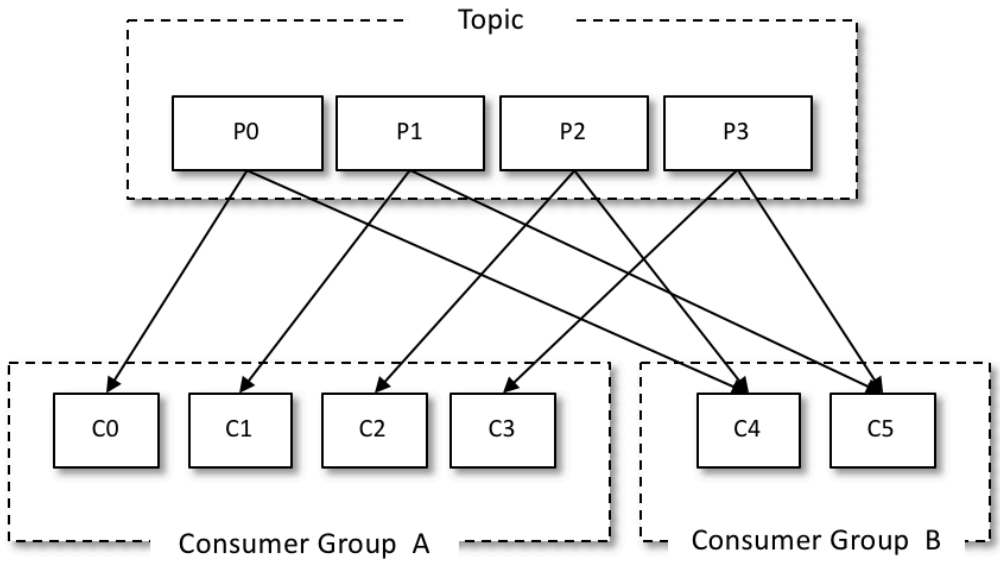
元数据的更新是由 Sender 线程负责的, 主线程也需要读取这些信息, 数据同步通过 synchronized 和 final 关键字来保障.

Outline

Kafka Consuming Model

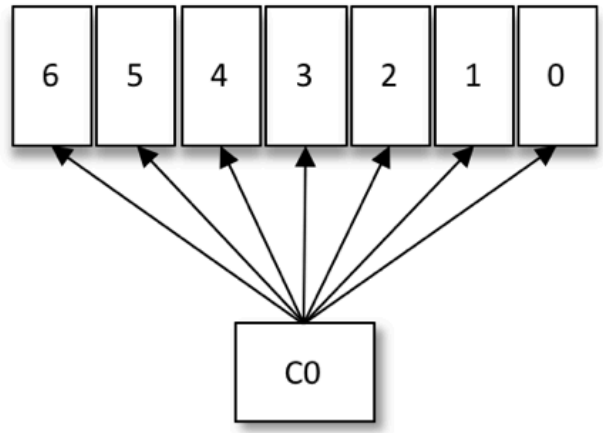
消费者与消费者组

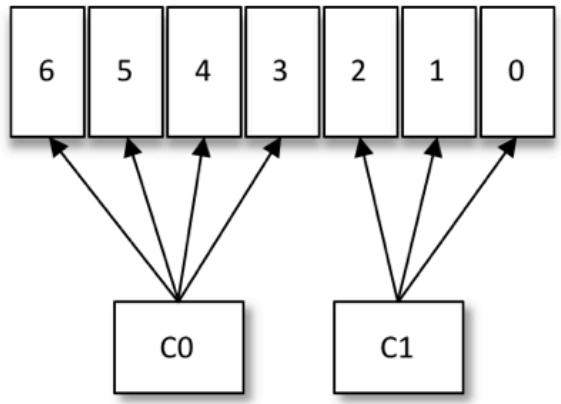
同一个分片只能由消费者分组中的同一个消费者进行消费



Kafka Consuming Model

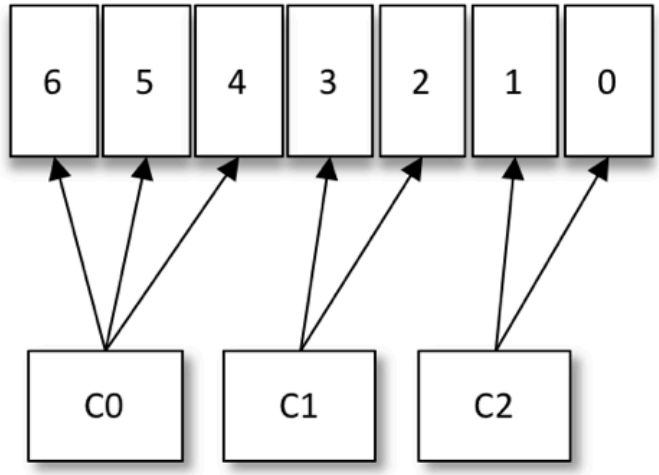
消费者与消费者组

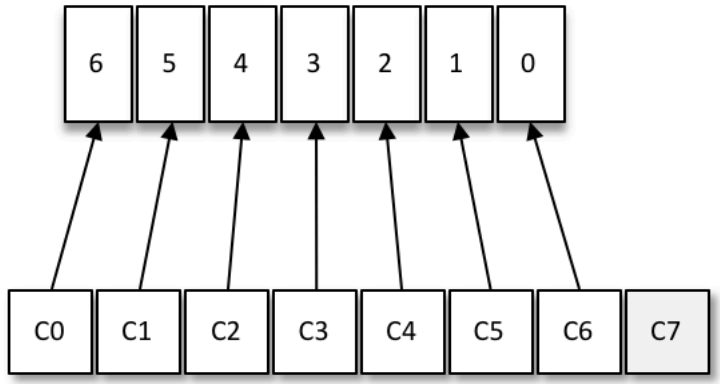




Kafka Consuming Model

消费者与消费者组





Kafka Consuming Model

消息消费

Kafka 中的消费是基于 pull 模式, 消费者主动向服务器发起请求拉取消息.

Kafka 中的消息消费是一个不断轮询的过程, 消费者重复调用 poll() 方法.

poll() 方法返回的是所订阅主题 (分区) 上的一组消息, 如果没有可供消费的消息, 则返回空集.

Kafka Consuming Model

Consumer 启动时消费位移

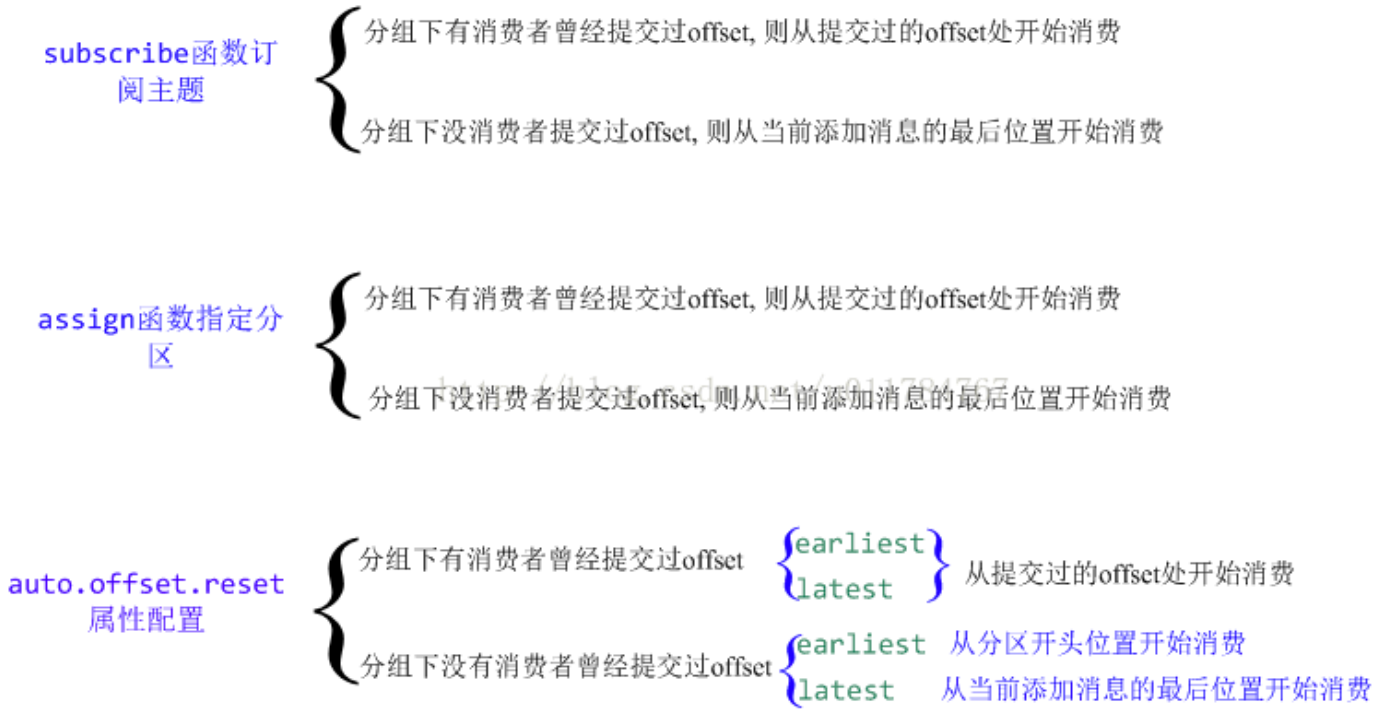


Figure 11: Consumer 启动时消费位移

Kafka Consuming Model

Consumer 启动时消费位移

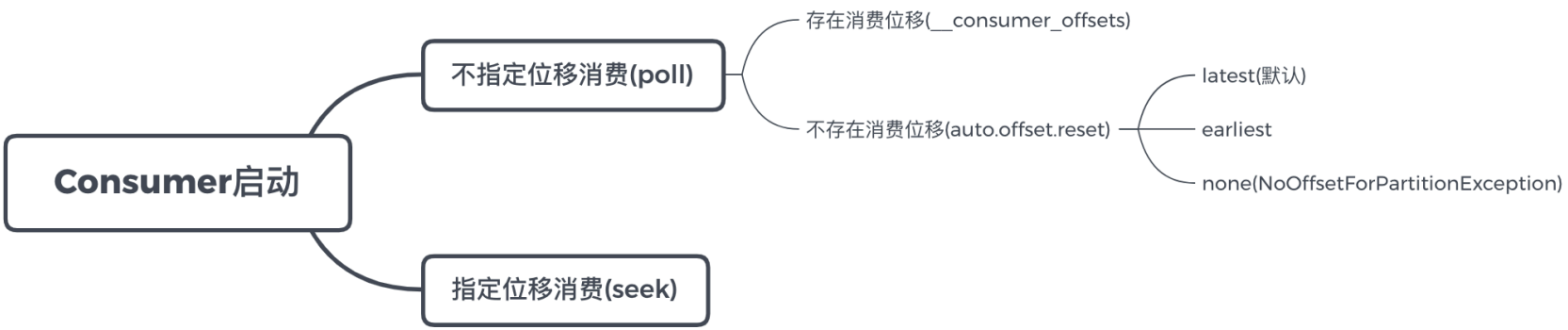


Figure 12: Consumer 启动时消费位移

Kafka Consuming Model

位移提交服务端

It has never changed from a external point of view, but internally, it did since Kafka 0.9, and the appearance of the __consumer_offsets.

A Group Coordinator is an Offset Manager(GroupMetadataManager) at the same time. It's a broker which is the leader for a group, that caches and commits its consumers offsets.

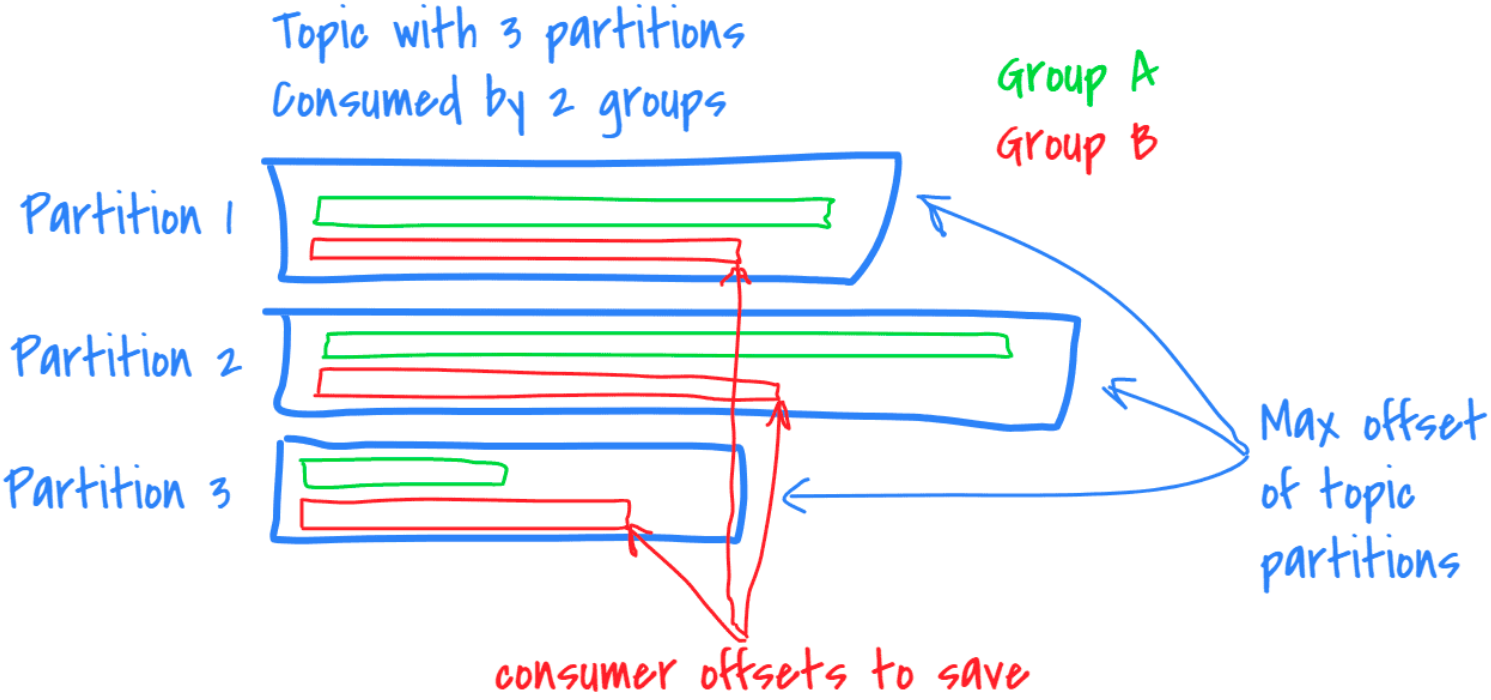
It's saved as binary data, each message in this topic has a key(group, topic, partition number) and a value.

```
1 [mygroup1,mytopic1,11]::[OffsetMetadata[55166421,NO_METADATA],CommitTime 1502060076305,ExpirationTime 1502146476305]
2 [mygroup1,mytopic1,13]::[OffsetMetadata[55037927,NO_METADATA],CommitTime 1502060076305,ExpirationTime 1502146476305]
3 [mygroup2,mytopic2,0]::[OffsetMetadata[126,NO_METADATA],CommitTime 1502060076343,ExpirationTime 1502146476343]
```

Figure 13: __consumer_offsets key:value

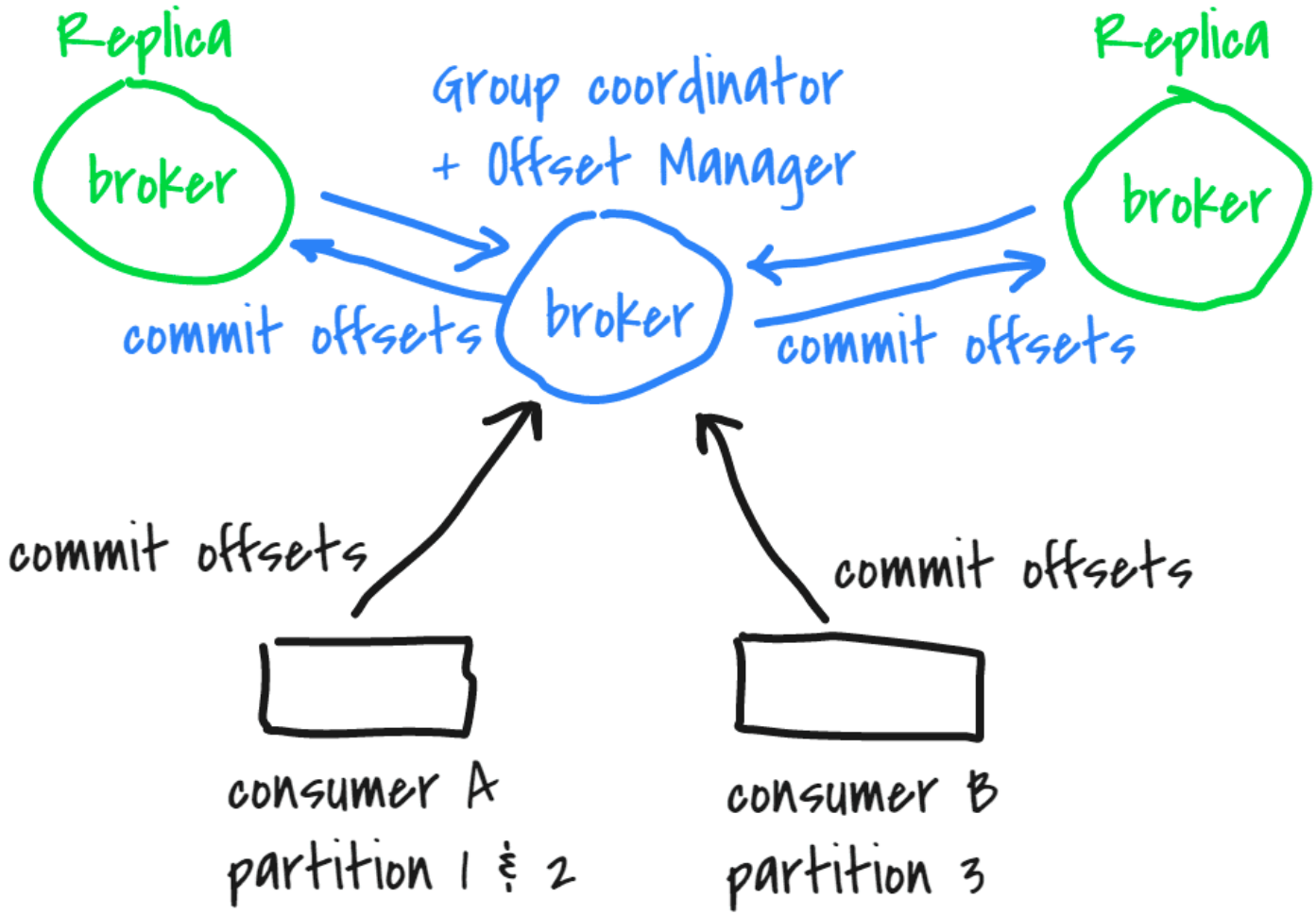
Kafka Consuming Model

位移提交服务端



Kafka Consuming Model

位移提交服务端



Kafka Consuming Model

位移提交服务端

```
[wangsheying@wangsheyingdeMacBook-Pro bin]$ ./kafka-run-class.sh kafka.admin.ConsumerGroupCommand \
> --bootstrap-server 10.41.17.57:9092 \
> --group kafka_test_01 \
> --describe

Consumer group 'kafka_test_01' has no active members.

GROUP           TOPIC            PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG             CONSUMER-ID     HOST            CLIENT-ID
kafka_test_01   test_scorecard_2 1          0               1               1               -               -               -
kafka_test_01   test_scorecard_2 0          0               1               1               -               -               -
kafka_test_01   test_scorecard_2 2          0               0               0               -               -               -
[wangsheying@wangsheyingdeMacBook-Pro bin]$
```

Figure 14: kafka-run-class.sh kafka.admin.ConsumerGroupCommand

Kafka Consuming Model

位移提交服务端

Conclusion

- __consumer_offsets is an implementation detail (came in 0.9) we should not rely on; it replaces the old system based on Zookeeper.
- __consumer_offsets is binary encoded. It keeps the latest consumed offsets for each topic/group/partition for a certain time only (1d, 过期自动删除).
- ConsumerGroupCommand can be used to retrieve the consumers offsets.
- There is one broker that deals with offset commits: the GroupCoordinator/OffsetManager.
- Low-level consumers can choose to not commit their offsets into Kafka (mostly to ensure at-least/exactly-once).

Kafka Consuming Model

位移提交客户端

位移提交是 Kafka 提供的一个语义保障，即如果你提交了位移 X，那么 Kafka 会认为所有位移值小于 X 的消息你都已经成功消费了。

位移提交的语义保障需要由Comsumer来负责，Kafka 只会“无脑”地接受提交的位移

- 自动提交
  - enable.auto.commit (默认 true)
  - auto.commit.interval.ms (默认 5s)
- 手动提交
  - 同步提交
  - 异步提交

Kafka Consuming Model

自动提交特性

Kafka 保证在开始调用 poll 方法时，提交上次 poll 返回的所有消息.

从顺序上来说，poll 方法的逻辑是先提交上一批消息的位移，再处理下一批消息.

- 保证不丢失
- 可能会重复

但是存在缓存池的情况下, 可能丢失.

Kafka Consuming Model

手动提交特性

- 同步提交
  - 阻塞
  - 自动重试
- 异步提交
  - 不阻塞
  - 不会重试 (位移值可能早已“过期”或不是最新值)

Kafka Consuming Model

手动提交特性

确保关闭消费者或再均衡前的最后一次提交成功

```
try {
    while (true) {
        ConsumerRecords<String, String> records = consumer.poll(Duration.ofSeconds(1));
        process(records);
        commitAysnc();
    }
} catch (Exception e) {
    e.printStackTrace();
} finally {
    try {
        consumer.commitSync();
    } finally {
        consumer.close();
    }
}
```

Figure 15: 同步异步组合提交

Kafka Consuming Model

手动提交特性

如果一次拉取了很多消息但是没有消费完，提交已经消费完成的位置

```
Map<TopicPartition,OffsetAndMetadata> currentOffset = new HashMap<>();
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records) {
        System.out.printf("offset = %d, key = %s, value = %s%n", record.offset(), record.key(),
record.value());
        currentOffset.put(new TopicPartition(record.topic(),record.partition()),new
 OffsetAndMetadata(record.offset(),"metadata"));
        try {
            System.out.println("模拟消息处理失败的情况");
        } catch (Exception e) {
            consumer.commitAsync(currentOffset,null);
        }
    }
}
```

Figure 16: 特定偏移量提交

Kafka Consuming Model

再均衡和多线程

再均衡是指分区的所有权从一个消费者转移到另一个消费者的行为.

- 消费组内增删消费者
- 再均衡期间, 消费组不可用 (时间短暂)
- 分区重新分配到的消费者, 当前状态丢失

KafkaProducer 是线程安全的, KafkaConsumer 是非安全的.

多线程消息消费最常见的方式是线程封闭, 即为每个线程实例化一个 KafkaConsumer 对象.

Kafka Consuming Model

Kafka 消费者重要参数配置

- fetch.min.bytes(1B)
- fetch.max.bytes(50MB)
- fetch.max.wait.ms(500ms)
- max.partition.fetch.bytes(1MB)
- max.poll.records(500 条)

Outline

分区管理

基本概念

Producer 和 Consumer 的设计理念所针对的都是主题和分区层面的操作

- 主题是消息的归类
- 分区是消息的二次归类
  - 分区提供水平扩展
  - 多副本提高数据可靠性
    -
    - 多副本提高数据可靠性

分区管理

分区管理

只有 leader 副本对外提供读写服务, follower 副本只负责在内部进行消息同步.

针对同一个分区而言, Kafka 集群中一个 broker 节点最多只能有一个副本.

Kafka 集群宕机或者添加新机器, 可能造成负载失衡, 可以通过优先副本再平衡或者分区重分配解决.

分区存储

逻辑层 VS 物理层

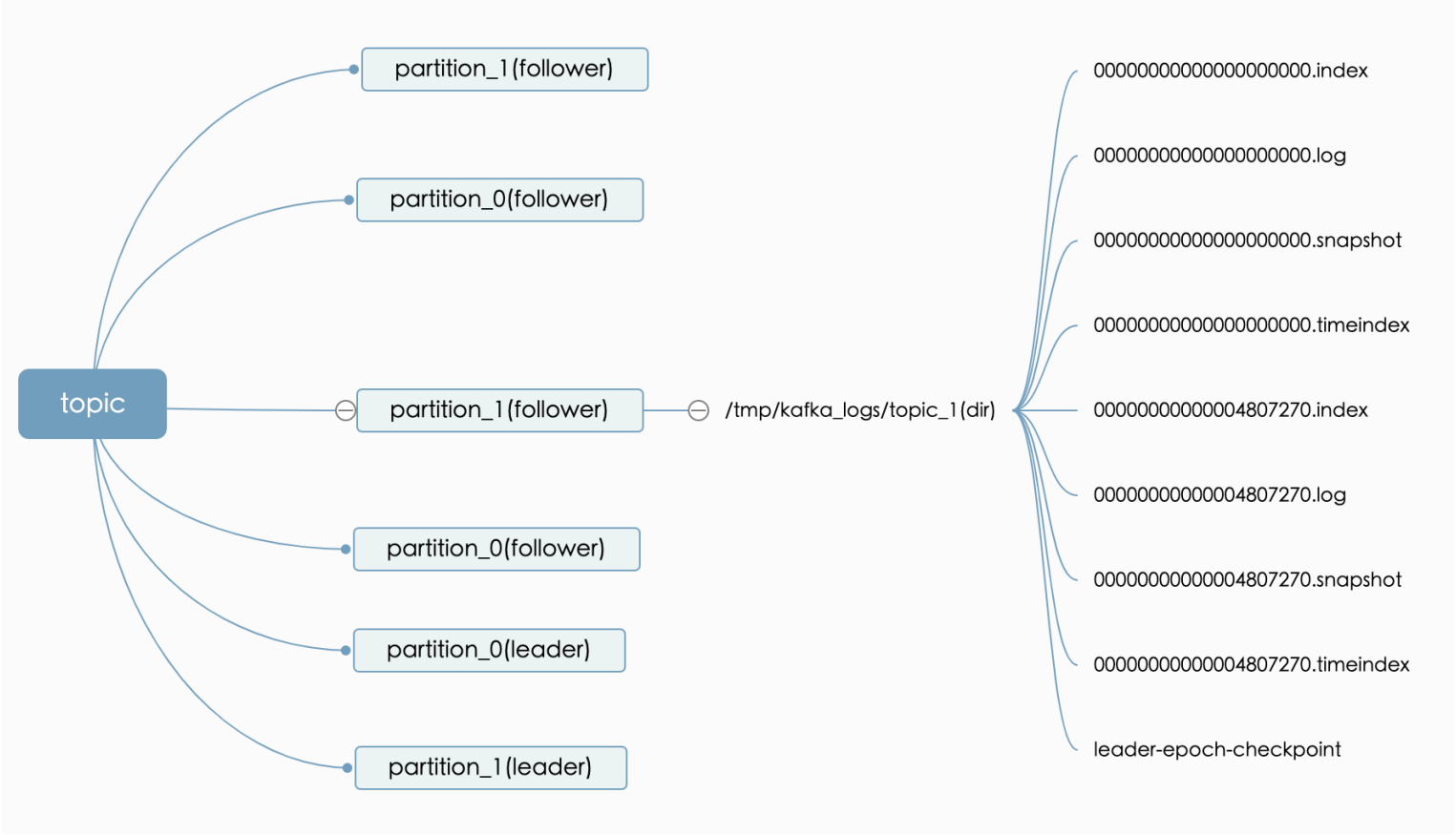


Figure 17: 逻辑层 VS 物理层

分区存储

基准偏移

每个 LogSegment 都有一个基准偏移量 baseOffset, 用来表示当前日志文件中第一条消息的 offset.

偏移量索引文件建立了消息偏移量 (offset) 到物理地址的映射关系; 时间戳索引文件类似.

Kafka 中的索引文件是以稀疏索引 (sparse index) 的方式构造消息的索引.

- log.index.interval.bytes(默认 4kb)
- MappedByteBuffer(Java NIO)

分区存储

文件目录布局

```
wangsheying@wangsheyingdeMacBook-Pro-2 kafka-logs % pwd
/tmp/kafka-logs
wangsheying@wangsheyingdeMacBook-Pro-2 kafka-logs % ls -lh
total 32
-rw-r--r--   1 wangsheying  wheel     0B  4 15 13:07 cleaner-offset-checkpoint
-rw-r--r--   1 wangsheying  wheel     4B  4 15 14:47 log-start-offset-checkpoint
-rw-r--r--   1 wangsheying  wheel    88B  4 15 14:33 meta.properties
-rw-r--r--   1 wangsheying  wheel    46B  4 15 14:47 recovery-point-offset-checkpoint
-rw-r--r--   1 wangsheying  wheel    46B  4 15 14:48 replication-offset-checkpoint
drwxr-xr-x  17 wangsheying  wheel   544B  4 15 14:36 topic_test-0
drwxr-xr-x  17 wangsheying  wheel   544B  4 15 14:36 topic_test-1
wangsheying@wangsheyingdeMacBook-Pro-2 kafka-logs % cd topic_test-1
wangsheying@wangsheyingdeMacBook-Pro-2 topic_test-1 % ls -lh
total 911248
-rw-r--r--  1 wangsheying  wheel   797K  4 15 14:33 00000000000000000000.index
-rw-r--r--  1 wangsheying  wheel   403M  4 15 14:23 00000000000000000000.log
-rw-r--r--  1 wangsheying  wheel    12B  4 15 14:33 00000000000000000000.timeindex
-rw-r--r--  1 wangsheying  wheel    16K  4 15 14:34 00000000000004807270.index
-rw-r--r--  1 wangsheying  wheel   8.0M  4 15 14:34 00000000000004807270.log
-rw-r--r--  1 wangsheying  wheel     0B  4 15 14:34 00000000000004807270.timeindex
-rw-r--r--  1 wangsheying  wheel    16K  4 15 14:35 00000000000004904329.index
-rw-r--r--  1 wangsheying  wheel   8.0M  4 15 14:35 00000000000004904329.log
-rw-r--r--  1 wangsheying  wheel    10B  4 15 14:34 00000000000004904329.snapshot
-rw-r--r--  1 wangsheying  wheel     0B  4 15 14:35 00000000000004904329.timeindex
-rw-r--r--  1 wangsheying  wheel    10M  4 15 14:35 00000000000005000749.index
-rw-r--r--  1 wangsheying  wheel   1.2M  4 15 14:36 00000000000005000749.log
-rw-r--r--  1 wangsheying  wheel    10B  4 15 14:35 00000000000005000749.snapshot
-rw-r--r--  1 wangsheying  wheel    10M  4 15 14:35 00000000000005000749.timeindex
-rw-r--r--  1 wangsheying  wheel     8B  4 15 14:33 leader-epoch-checkpoint
wangsheying@wangsheyingdeMacBook-Pro-2 topic_test-1 %
```

Figure 18: 文件目录布局

分区存储

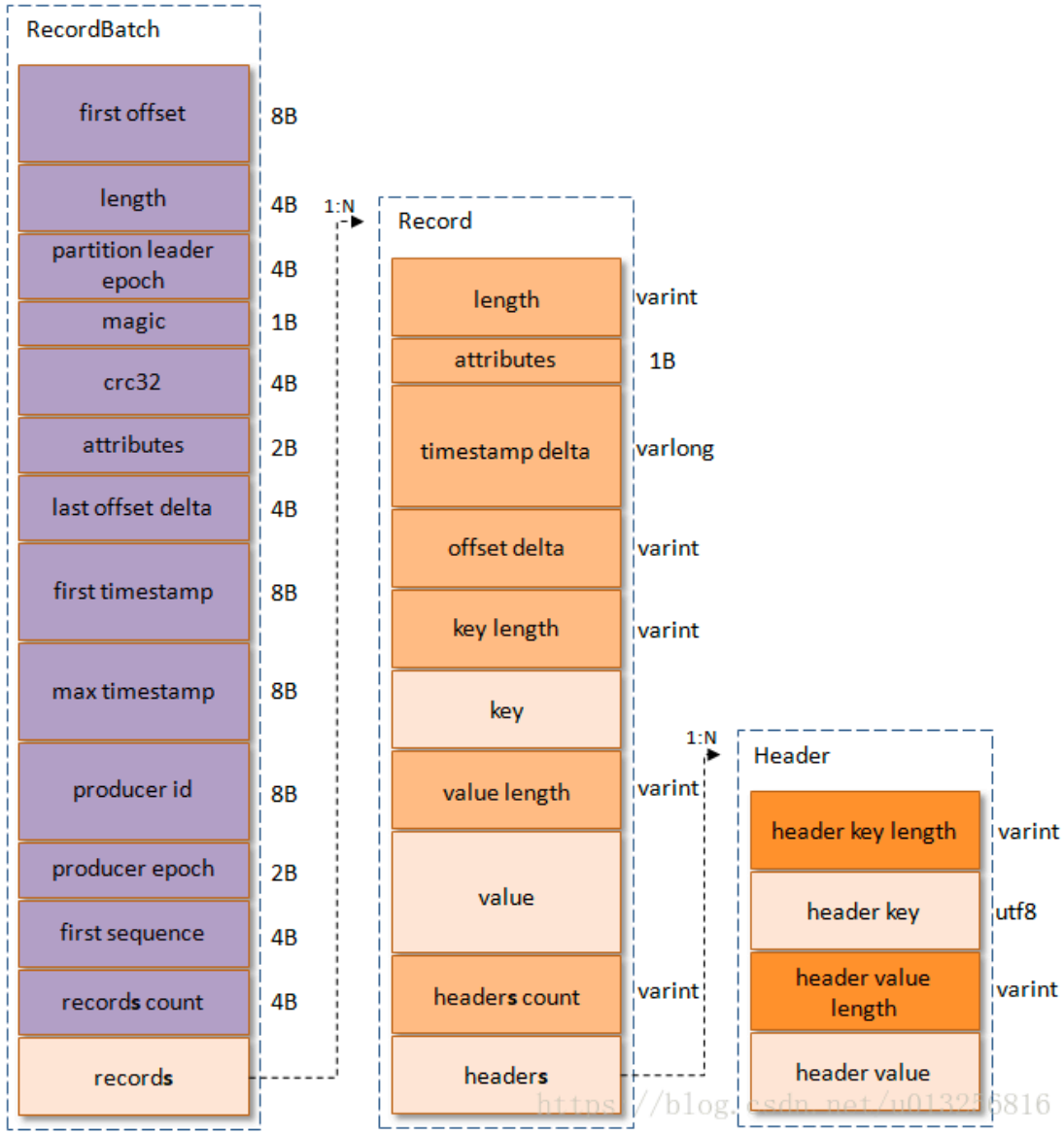
消息压缩

在一般情况下, 生产者发送的压缩数据在 broker 中也是保持压缩状态进行存储的, 消费者从服务端获取的也是压缩消息, 消费者在处理消息之前才会解压消息, 这样保持了端到端的压缩.

- gzip
- snappy
- lz4
- zstd
- uncompressed
- producer(默认)

分区存储

V2 版本消息格式 (v0.11.0+)



Kafka 为什么这么快

多因素叠加

- 文件分段 (针对 Producer,Consumer)
- 顺序 IO(较快, 针对 Producer,Consumer)
- 日志格式 (针对 Producer,Consumer)
- page cache(主要针对 Producer)
- 零拷贝 (极快, 仅针对 Consumer)
- 消除 JVM GC

Kafka 为什么这么快

磁盘速度

As a result the performance of linear writes on a JBOD configuration with six 7200rpm SATA RAID-5 array is about 600MB/sec but the performance of random writes is only about 100k/sec—a difference of over 6000X.

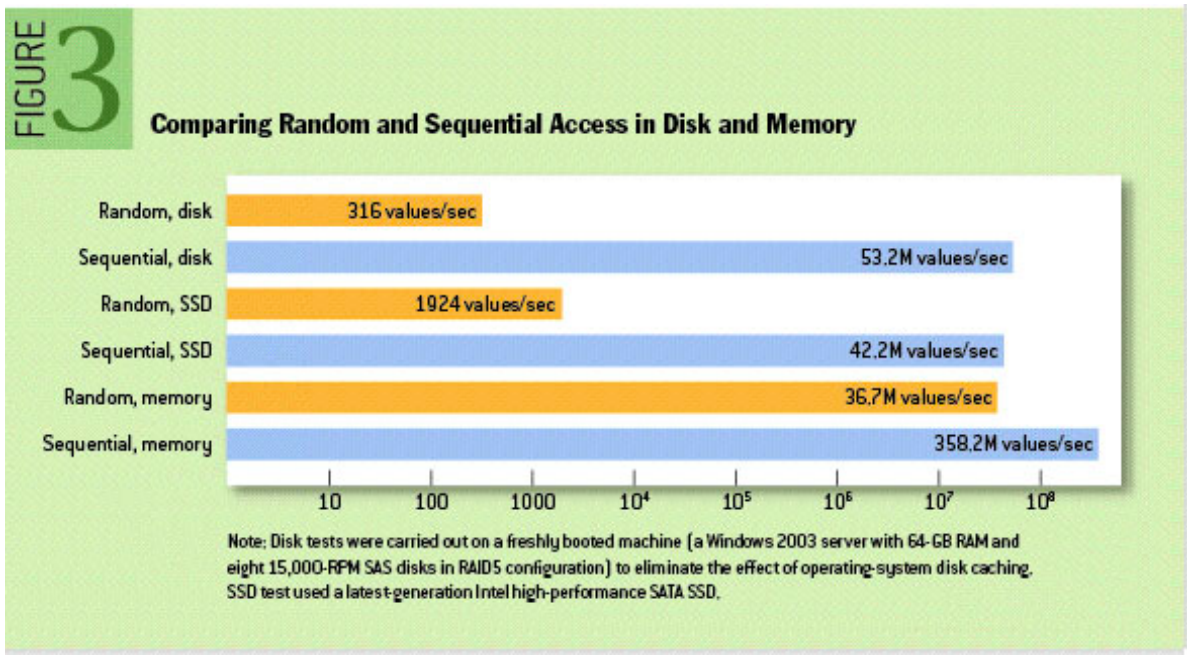


Figure 19: disk vs memory

Kafka 为什么这么快

zero copy

There are 4 context switches and 2 unnecessary copies.

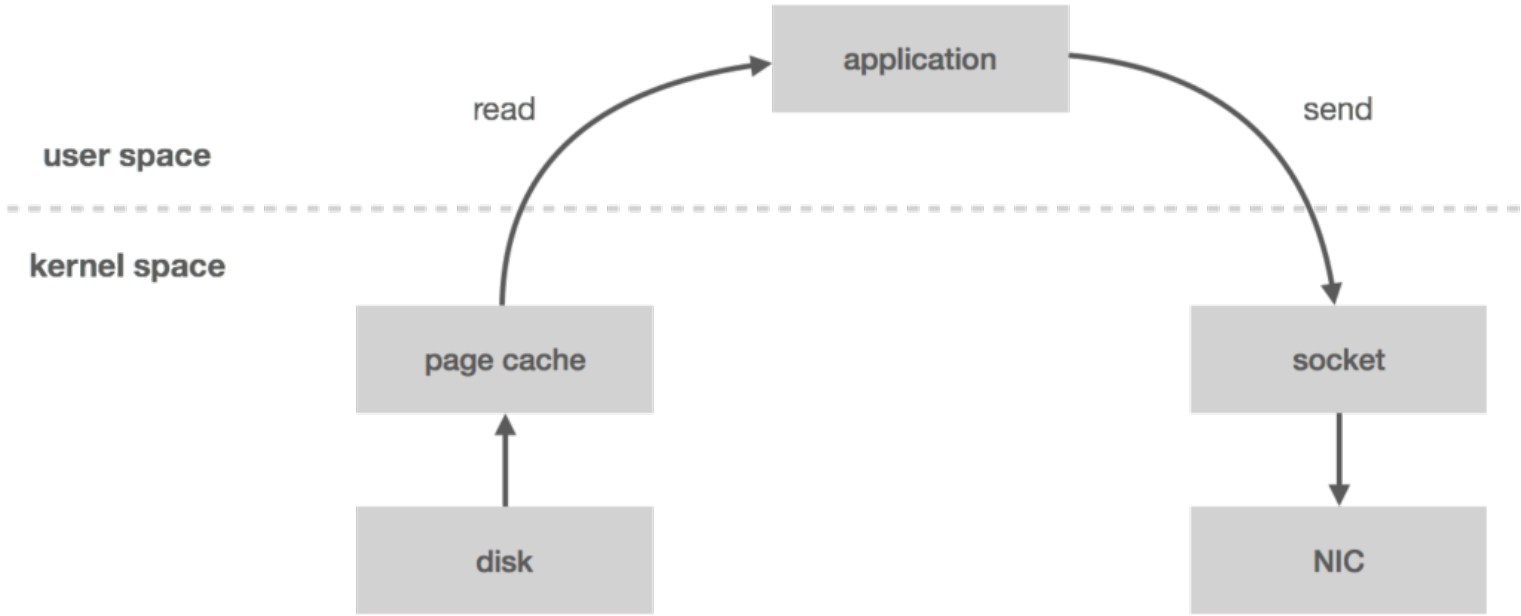


Figure 20: 内核态到用户态拷贝

Kafka 为什么这么快

zero copy

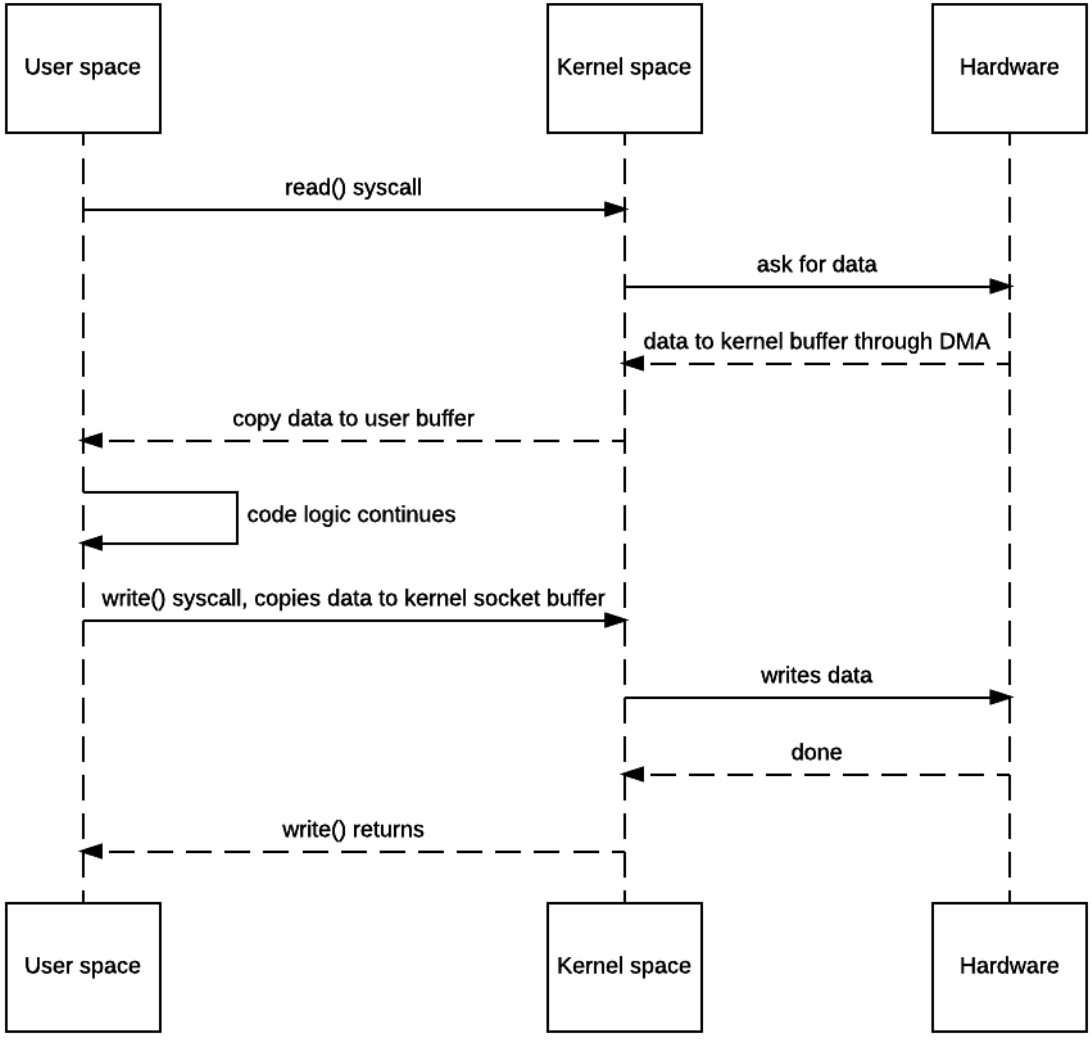


Figure 21: 内核态到用户态拷贝

Kafka 为什么这么快

zero copy

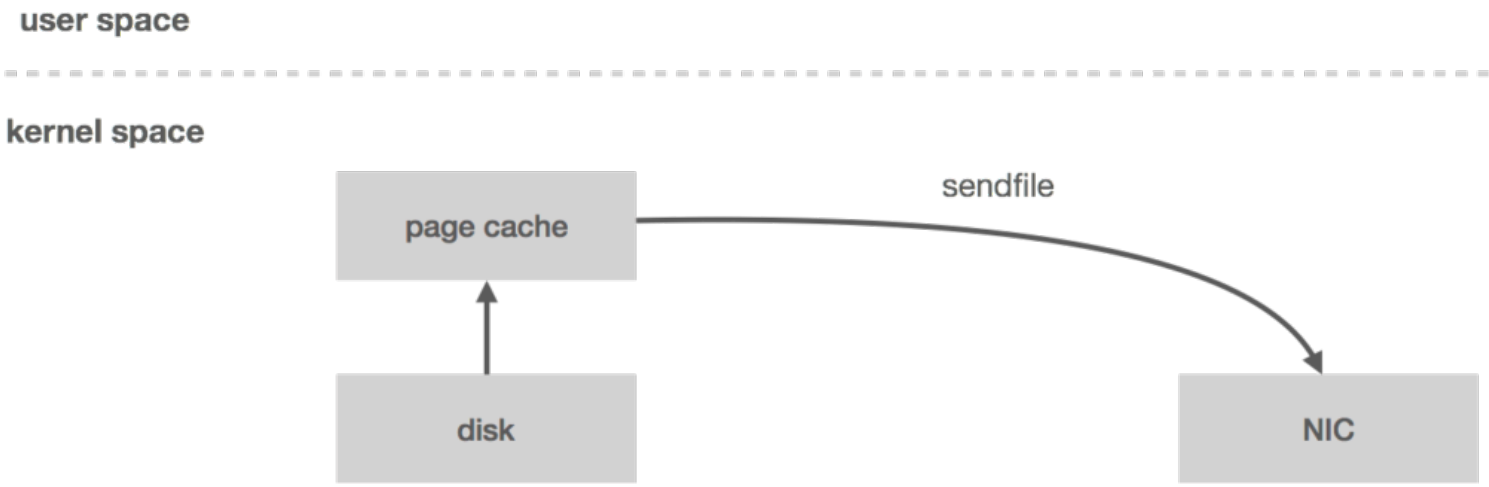


Figure 22: zero copy

Kafka 为什么这么快

zero copy

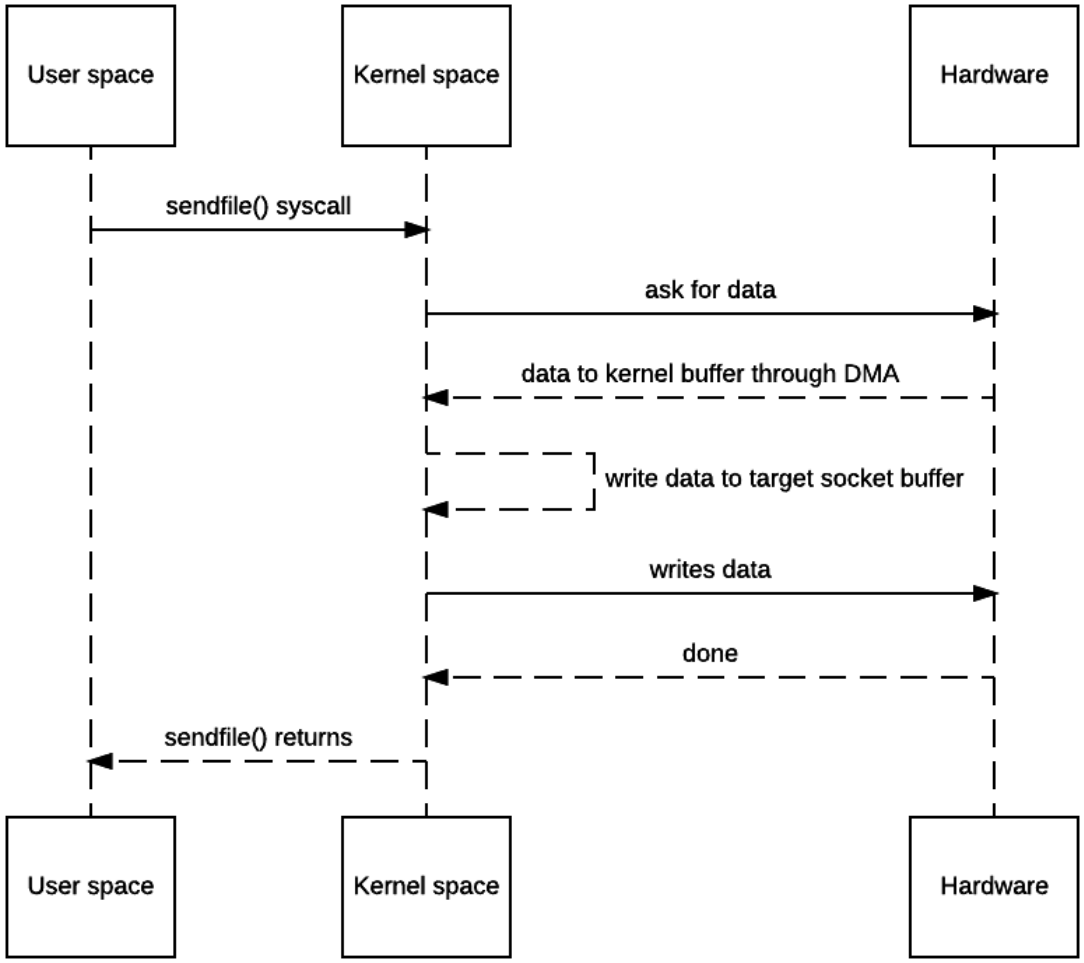


Figure 23: zero copy

Kafka 为什么这么快

消除 JVM GC

Furthermore, we are building on top of the JVM, and anyone who has spent any time with Java memory usage knows two things:

- The memory overhead of objects is very high, often doubling the size of the data stored (or worse).
- Java garbage collection becomes increasingly fiddly and slow as the in-heap data increases.

Using the filesystem and relying on pagecache will result in a cache of up to 28-30GB on a 32GB machine without GC penalties.

Outline

协议设计

基本概念

协议设计

基本概念

协议设计

基本概念

协议设计

基本概念

Outline

基本概念

基本概念

Outline

基本概念

基本概念

Questions and Answers?

Questions and Answers?

Thank You!

References I