Kafka原理及使用场景

黄康正

1. 什么是Kafka?

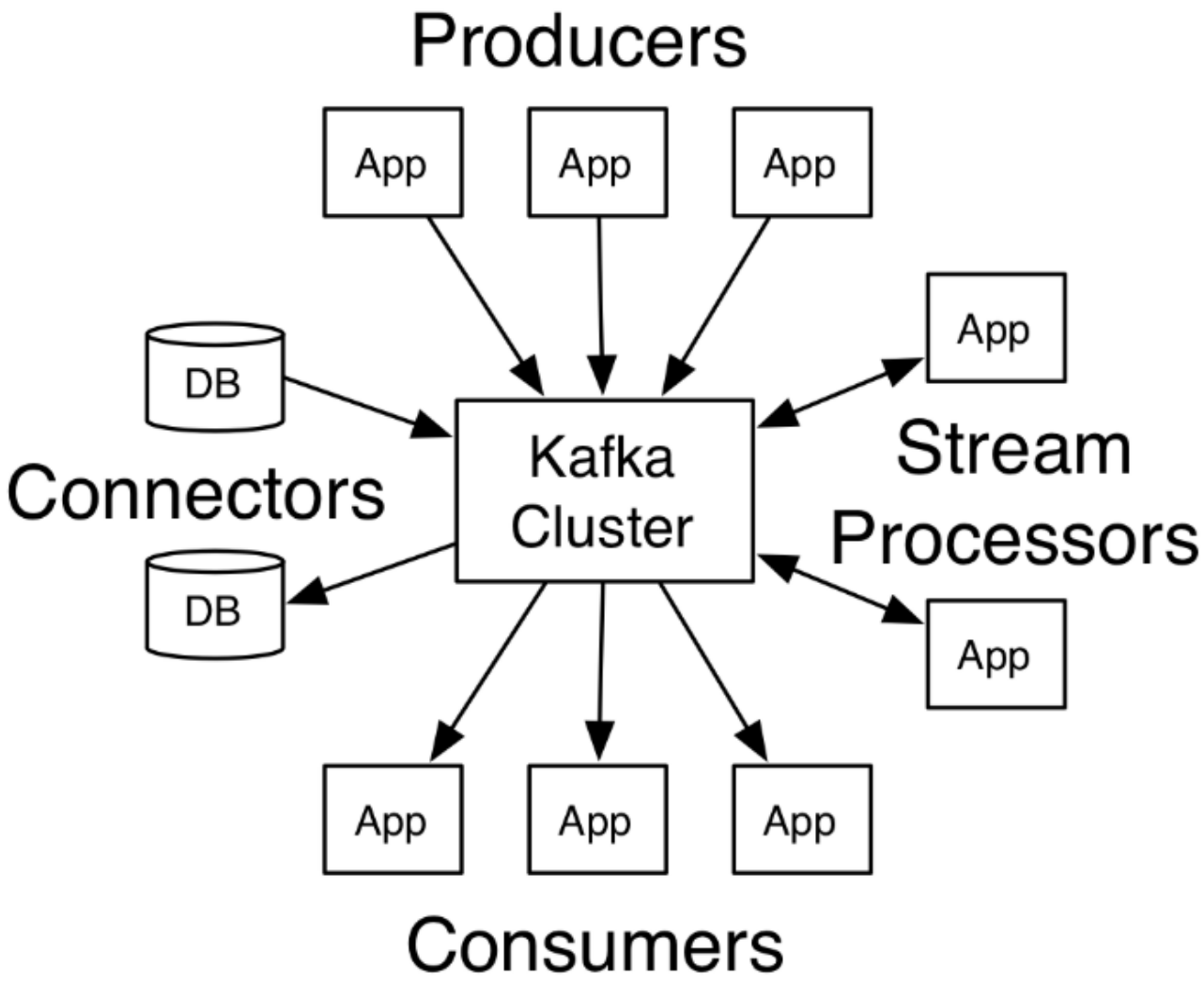
• 发布-订阅的消息系统

• 有容错机制的消息存储系统

• 流式处理平台

• 分布式，高吞吐量，水平拓展

2. 核心API



Kafka核心API图

3. 核心概念与原理剖析

3.1. Broker

Kafka节点，一个Kafka节点就是一个broker，多个broker可以组成一个Kafka集群

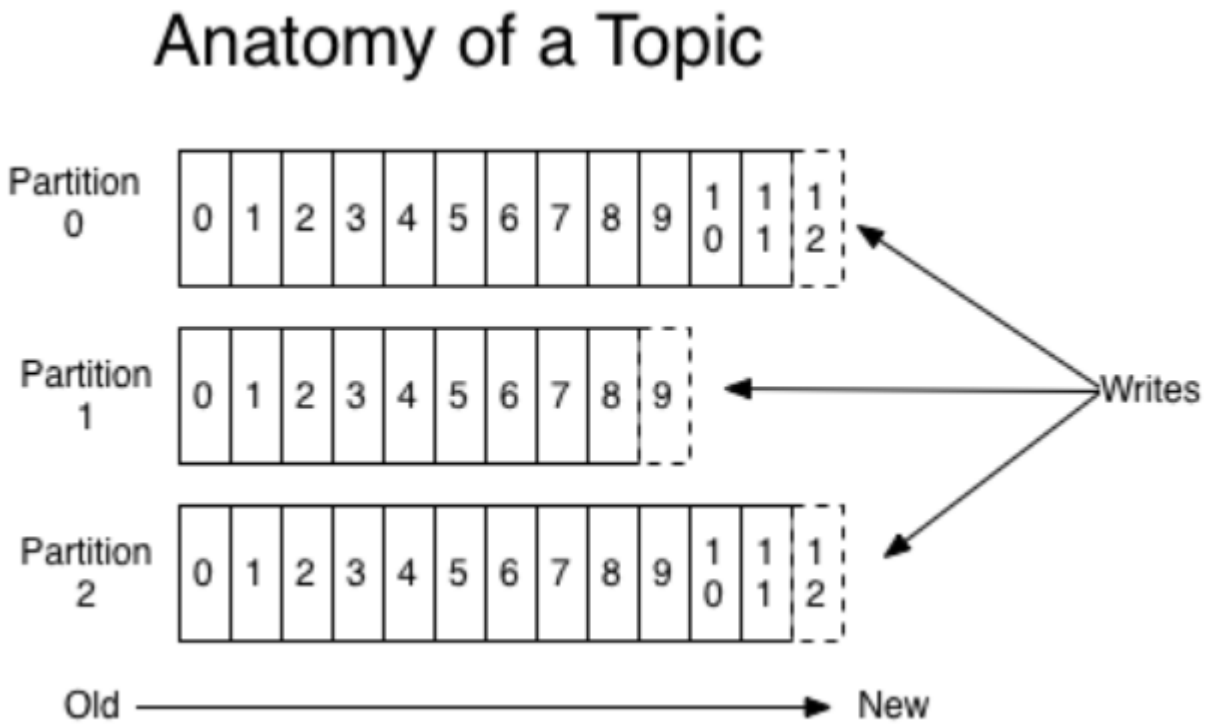
3.2. Topic, Partition, Segment, Logs

Topic: 消息被发布的类别，或者说是目录，类似mq中的topic

Partition: Topic物理上的分组，一个topic可以分为多个partition，每个partition是一个有序的队列

Segment: partition是分段存储的，每个段是一个segment

Logs: kafka中的记录存为log文件



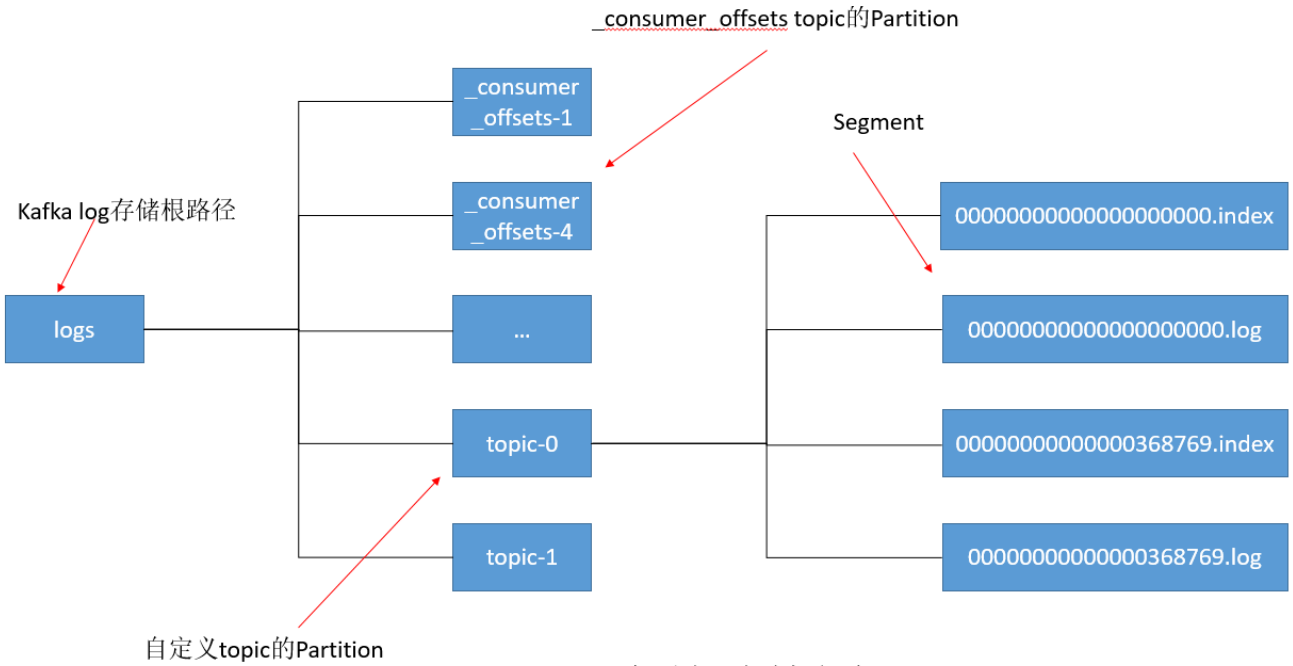
Topic 结构图

kafka使用顺序I/O，对日志文件进行append操作，因此磁盘检索的开支是较小的

同时为了减少磁盘写入的次数，broker会将消息暂时buffer起来，当消息的个数(或尺寸)达到一定阀值时，再flush到磁盘，这样减少了磁盘IO调用的次数

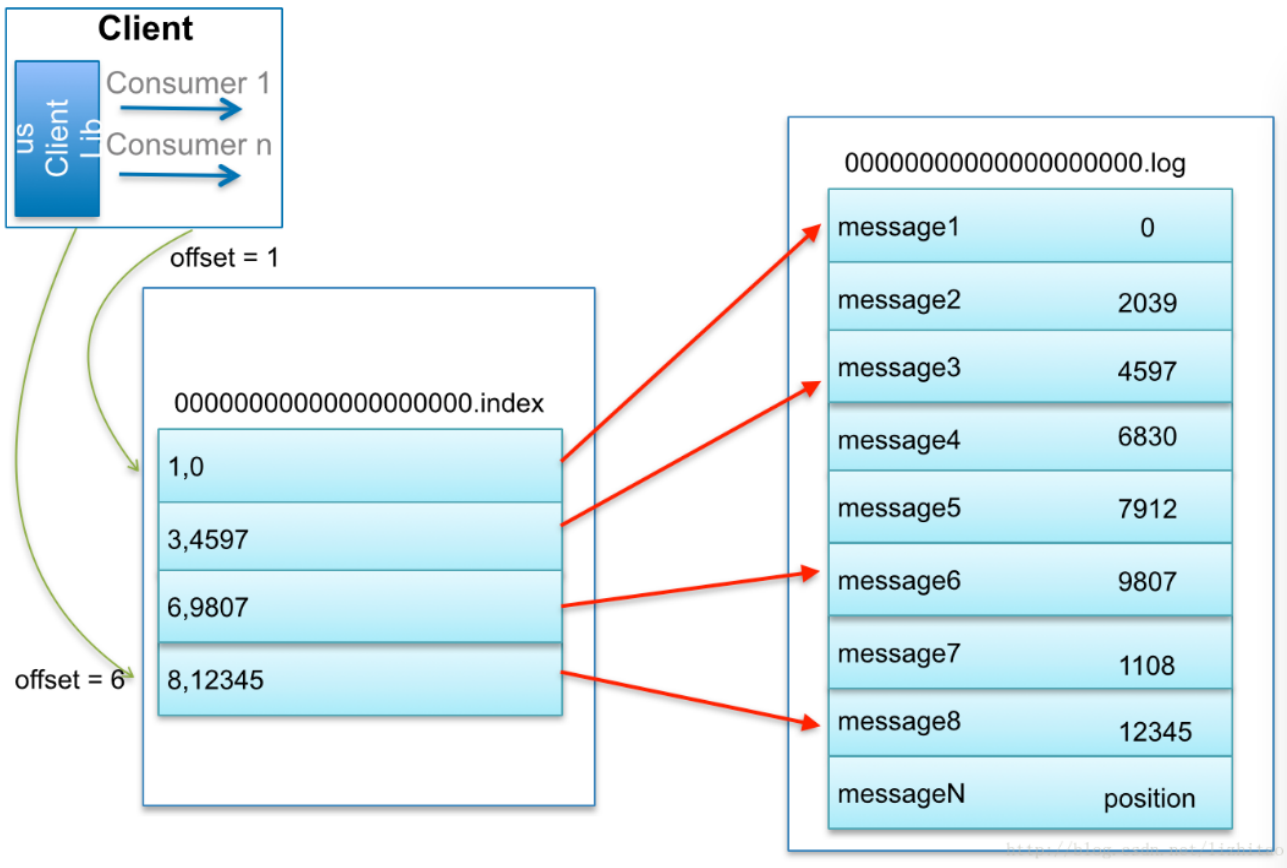
对于kafka而言，较高性能的磁盘，将会带来更加直接的性能提升

Kafka log中主要存储的有index（索引）以及log（记录）文件



Kafka存储结构图

下图演示了Consumer通过offset找到具体某条记录的详细过程



Log查找记录详解图

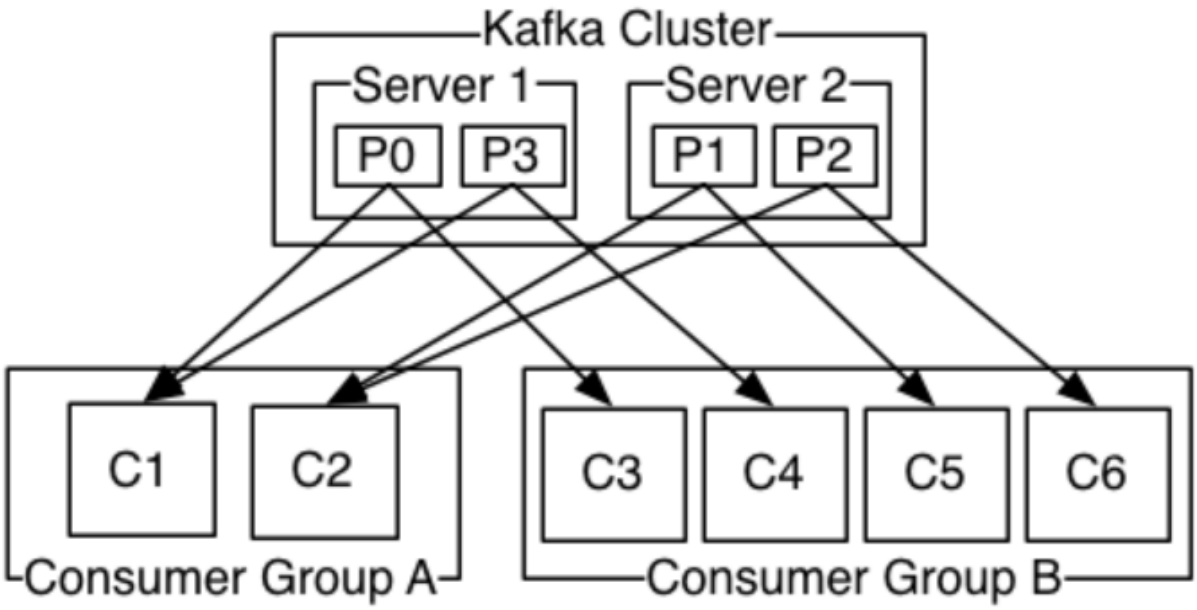
3.3. Producer

生产message发送到topic，可以指定partition生产

3.4. Consumer and Consumer Group

Consumer: 订阅topic消费message，consumer作为一个线程来消费，可以指定partition消费

Consumer Group: 一个Consumer Group包含多个consumer，Topic中的同一条数据只能由同个Consumer Group下的一个Consumer消费，Consumer数量小于等于Partition数量



Kafka Partition与Consumer的关系图

Rebalance: 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

3.5. Zookeeper

zookeeper是一个分布式服务框架，维护一个类似于文件系统的数据结构，目录节点发生变化会通知客户端

关于zookeeper选举与一致性等问题本文暂不详述，可以自行搜索paxos，quorum，ZAB协议等等

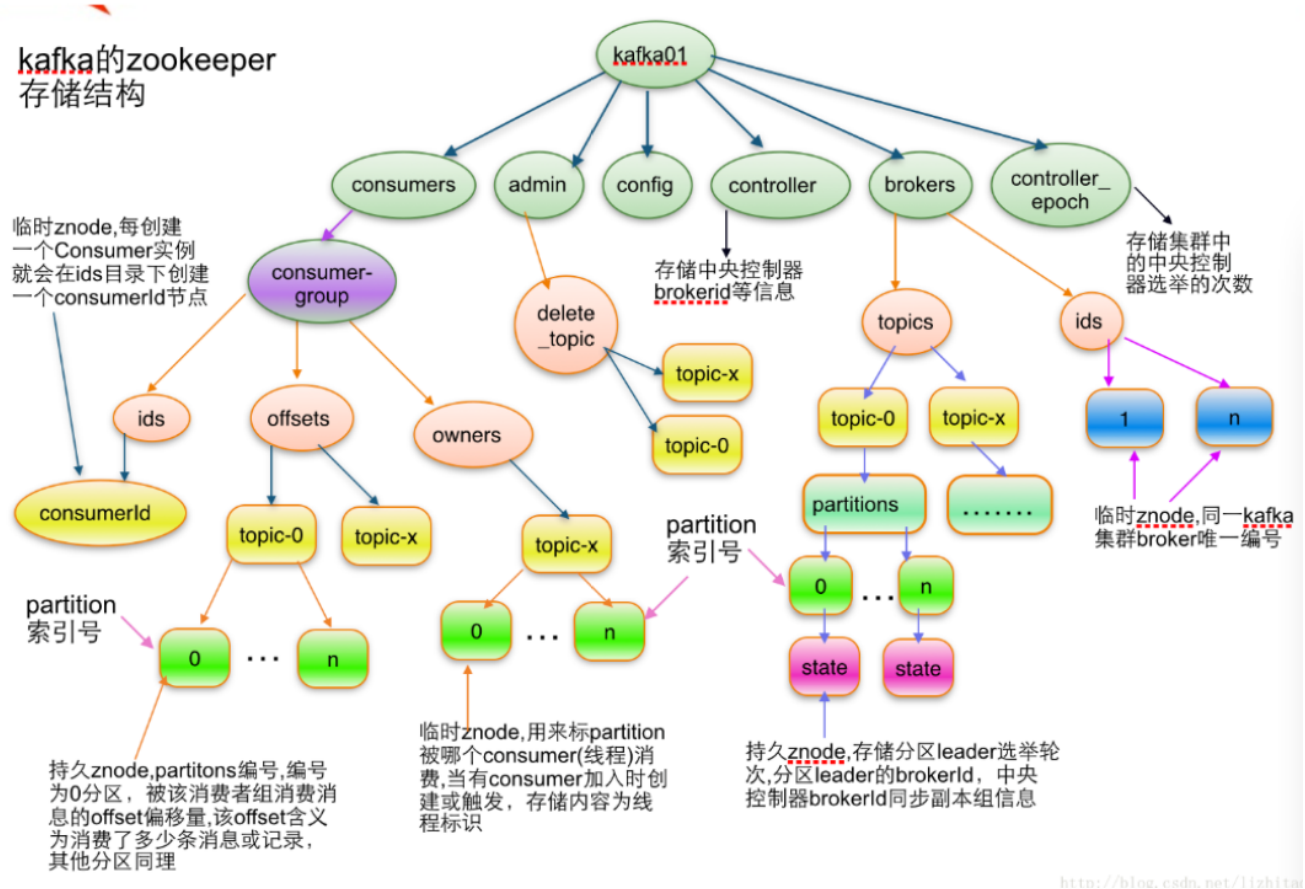
四种节点类型：PERSIST, PERSIST_SEQUENTIAL, EPHEMERAL, EPHEMERAL_SEQUENTIAL

（1）PERSIST：持久节点，会被持久化到磁盘之中，即使zookeeper重启之后，节点还是会存在

（2）EPHEMERAL：临时节点，zookeeper重启之后，不会存在；假设Client没有心跳了，这个节点也不会存在；Client与zookeeper的session结束了，这个节点也不会有了

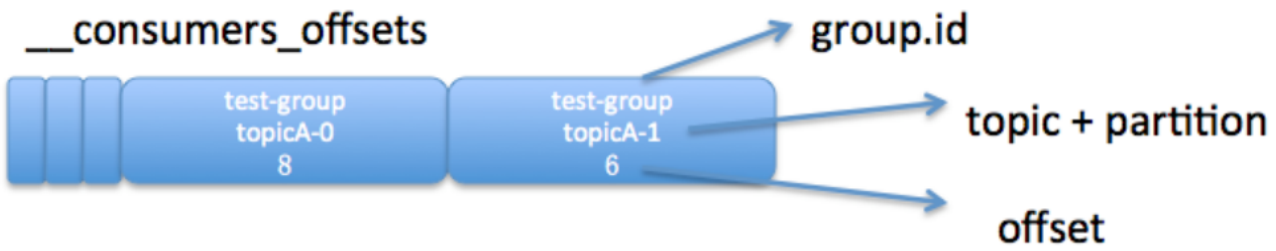
（3）SEQUENTIAL：顺序节点，假设不是顺序节点的话，Client1创建了节点/a，那么其它Client就不能再创建节点/a，否则将报错。但是如果是SEQUENTIAL节点的话，Client1创建了节点/a，若其它Client也创建节点/a，则不会报错，也可以创建成功，只不过命名规则会在末尾加序号，越早创建的序号靠前

Kafka使用zookeeper管理集群



kafka在zookeeper中的目录结构图

注：新版本的kafka已经不将consumer的offset存在zookeeper了，由于zookeeper并不适合进行大批量的写操作，因此kafka提供了另一种解决方案：增加__consumer_offsets topic，将offset信息写入这个topic，保存了每个consumer group某一时刻提交的offset信息，默认情况下有50个 partition



_consumers_offsets topic中的存储结构图

4. kafka集群

4.1. Leader Election

zookeeper 2种选举方式：抢注leader节点——非公平模式；先到先得，后者监视前者——公平模式

非公平模式：

（1）抢注/controller节点，谁先创建节点成功谁是leader

（2）节点应该设置成EPHEMERAL

（3）节点创建前会先查询是否存在/controller节点，存在则放弃创建，同时在/controller节点注册watch

（4）Leader宕机/controller节点会自行删除，其他节点会收到删除的通知，开启新一轮抢注

公平模式：

（1）先创建节点成功的id号较小，如/controller/0，后创建的依次序号增加

（2）后创建的节点watch前一个节点

（3）节点应该设置成EPHEMERAL_SEQUENTIAL

（4）创建节点成功后，调用getChildren获得/controller下的所有节点，如果节点id最小，则为leader

（5）Leader宕机节点删除，下一个节点接收到通知，重新进行第（4）步操作

Kafka中的leader election采取的是第一种方式

若是各个partition都通过zookeeper进行选举leader的话，zookeeper负载很大，十分耗性能

所以kafka的策略是：利用zookeeper选举出controller，通过controller指定partition的leader和follower

4.2. 高可用

在kafka中，leader负责读写，replica作为备份，从leader pull数据

当leader宕机，controller选取其他replica成为leader，即可继续使用

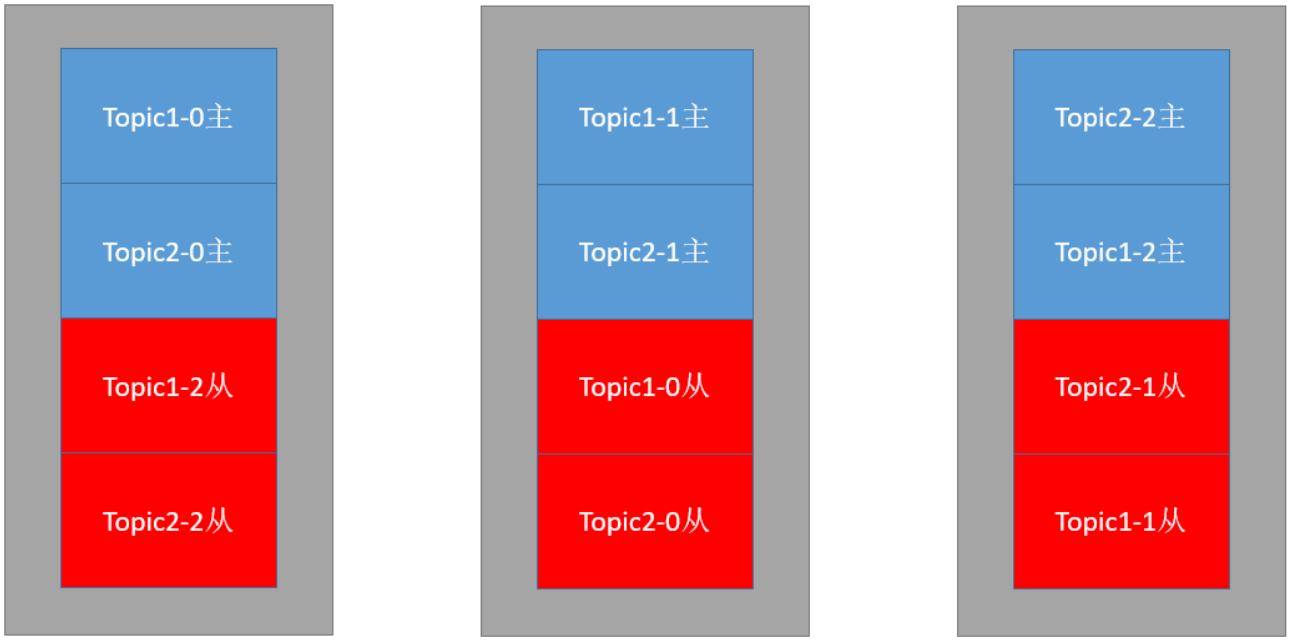
N个kafka broker宕机N-1个仍然可以使用

Kafka会将partition与replicas尽量均匀的分布在集群上

Kafka中分配partition与replica的算法：

（1）将第i个Partition分配到第（i mod n）个Broker上

（2）将第i个Partition的第j个Replica分配到第（(i + j) mod n）个Broker上



Partition在Broker的存储结构图

思考：为什么kafka不设计成follower也能读数据?

follower可以读数据，意味着consumer可以从follower消费数据，会有数据一致性问题，而解决数据一致性需要考虑同步，同步影响性能，kafka的设计目标就是为了大数据量，高吞吐量的业务场景

4.3. 主从同步

同步复制VS异步复制：

同步复制：所有的follower复制好数据后才commit，一致性高，但是效率低

异步复制：leader拿到数据立刻commit，follower后台慢慢异步复制，效率高，但一致性低，follower没复制成功leader就宕机的话，follower就没有这条数据了

ISR：

（1）leader会维护一个与其基本保持同步的Replica列表，该列表称为ISR(in-sync Replica)，每个Partition都会有一个ISR，而且是由leader动态维护

（2）broker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

（3）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将其从ISR中移除

（4）ISR中的Replica才有资格成为leader

（5）ISR存储路径：/broker/topics/{topic}/partitions/{partition}/state

```
##zookeeper中存储的partition的replica信息
get /brokers/topics/testTopic

{"version":1,"partitions":{"2":[2,0],"1":[1,2],"0":[0,1]}}


##zookeeper中存储的partition的ISR信息
get /brokers/topics/testTopic/partitions/0/state

{"controller_epoch":2,"leader":0,"version":1,"leader_epoch":0,"isr":[0,1]}
```

5. 消息中间件的通性问题——消息丢失，消息重复?

消息投递的三种语义：

At Most Once: 最多一次，不需要ack，也没有fail重传机制，只管发了就结束了

优点是绝对不会重复传输；缺点是消息容易丢

At Least Once: 最少一次，有ack和fail机制，需要接受到ack才发送成功，fail的话会重传，直到接收到ack

优点是消息不容易丢；缺点是可能会重复传输

Exactly Once：精确一次，确保receiver只接收到一次

幂等性：一次和多次请求某一个资源**对于资源本身**应该具有同样的结果

5.1. Producer

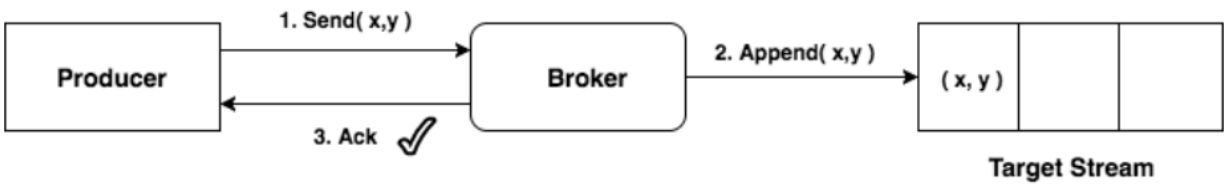
5.1.1. Ack

request.required.acks参数的设置来进行调整：

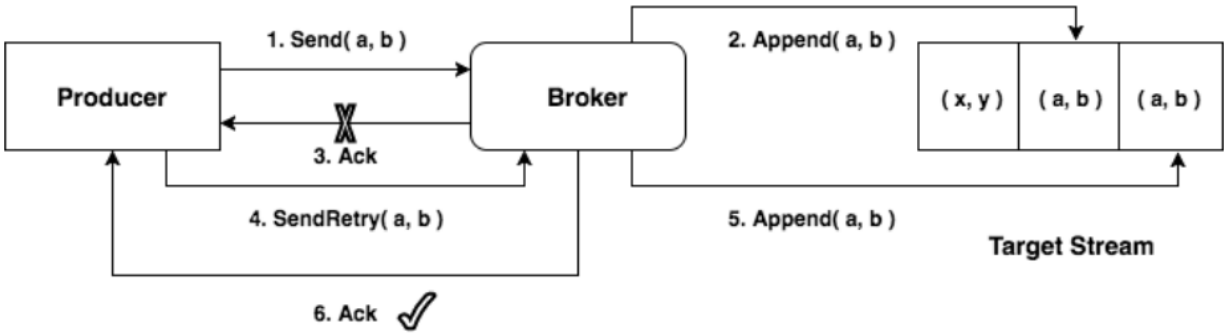
0，相当于异步发送，不需要leader回复，消息发送完毕即offset增加，发送成功继续生产；相当于At most once

-1，ISR列表中的所有replica都同步消息成功返回了ack，leader才确认接收到消息，可靠性高，但性能较低（假设leader宕机了，follower顶上去还是最新的数据）

1，leader接收到消息后就返回ack，replica后台同步消息，中和可靠性与性能（假设leader ack了消息，然后follower还没有同步到消息leader就宕机了，这个消息就丢失了）



Favorable Case :)



Gone Case :(

Acknowledgement failed, led to message duplication

消息重复消费场景图

5.1.2. 幂等Producer

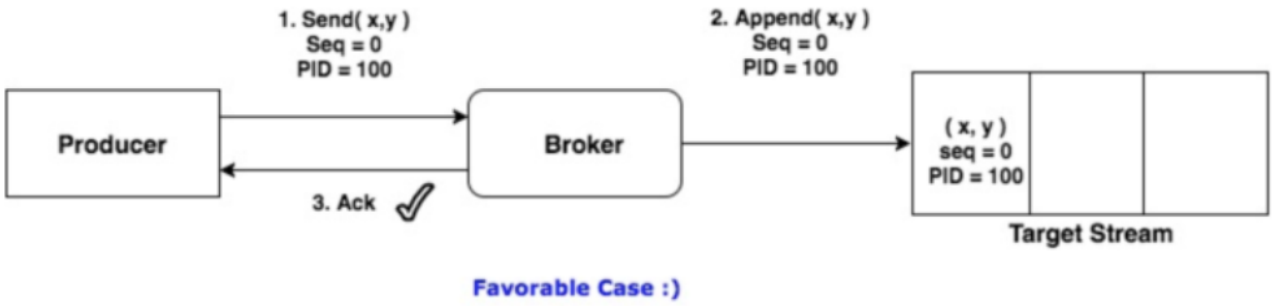
enable.idempotence参数设置，设置为true

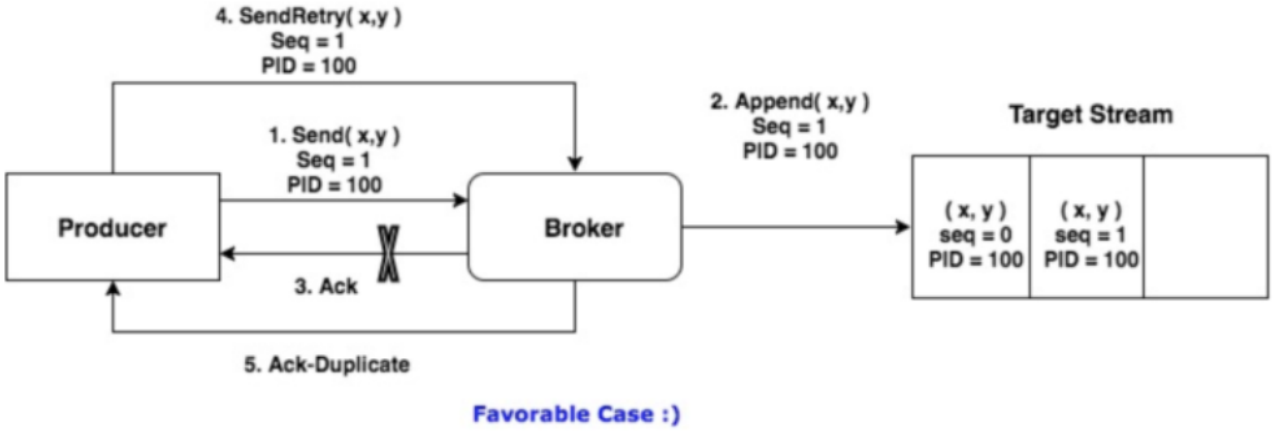
关于Kafka幂等Producer的实现：

为了实现Producer的幂等性，Kafka引入了Producer ID（即PID）和Sequence Number

- PID：每个新的Producer在初始化的时候会被分配一个唯一的PID，这个PID对用户是不可见的
- Sequence Numbler：（对于每个PID，该Producer发送数据的每个<Topic, Partition>都对应一个从0开始单调递增的Sequence Number

Broker端保存了这seq number，对于接收的每条消息，如果其序号比Broker中序号大于1则接受它，否则将其丢弃，这样就可以实现了消息重复提交





幂等Producer重复消费场景图

5.1.3. Kafka事务

幂等Producer保证了对于同一个patition的重复写入，只会写入一次，但无法保证对于同一个Producer对于一个topic的不同partition的幂等性，而要解决这个问题就需要引入事务的概念

类似与数据库事务，生产者多次发送消息可以封装成一个原子操作，要么都成功，要么失败

应用场景：

Producer生产多条消息：需配置transactional.id（可理解为事务名称）属性与enable.idempotence属性

消费-生产并存（Consume-transform-produce）：消费者需要将auto.commit设置为false，设置isolation.level（理解成事务隔离级别）

5.2. Consumer

PUSH VS PULL：

Push：Broker推送给consumer，优点是实时性好，缺点是broker很难控制数据发送给消费者的速度

Pull：consumer主动从broker拉取消息，优点是消费速率可控，缺点是实时性低

在Kafka中consumer消费数据是Pull模型

思考：为什么kafka中consumer是pull模型？

5.2.1. Consumer幂等性问题

产生原因：

（1）由于消费者消费消息之后，来不及向系统提交offset数据，有可能这个时候，系统宕机，此时zookeeper中存放的offset的数据是上一次提交的数据，所以不是最新的offset

（2）消费者如果不是消费一条就提交一次offset，而是批量的提交offset，如果在提交之前系统宕机，也是会导致zookeeper中存放的数据不是最新的offset

解决：

kafka不做consumer幂等处理

（1）将offset存在应用或者第三方存储中（如redis），消费消息时判断是否有这条数据，有则不做处理

（2）业务处理，业务端判断消费到了重复的数据（如id唯一，业务端判断id重复），则不做处理

Kafka consumer消费数据很自由，可以任意选择offset去拉数据，主要看场景

6. 应用场景与总结

其实到这边kafka的各种特性已经理解的很深刻了，高吞吐量，高并发，高可拓展性，消息持久化存储，高可用等等，其设计目标与特性为：

（1）生产消费数据快（读写文件快，时间复杂度为O(1)的磁盘访问能力，时间复杂度O(1)的消息持久化能力）

（2）高吞吐量，Kafka可以提供单机每秒10万次的消息处理能力

（3）消息分区存储，分布式消费，partition内有序，消费快（多个consumer消费）可用性高

（4）同时支持数据离线处理和数据实时处理

虽然其也支持了消息中间件的特性（如消息丢失与重复问题的处理等等），但其主要设计目标还是在于大数据量的场景，以及流式处理大数据等等，其设计之初就是一个处理log的分布式系统，且其核心特性就是高吞吐量，所以其应用场景主要就是应用在大数据，日志系统，流式处理平台（类似apache storm）等等上

拓展话题——各大消息中间件特性，我该如何选择？