# kafka

# Kafka简介

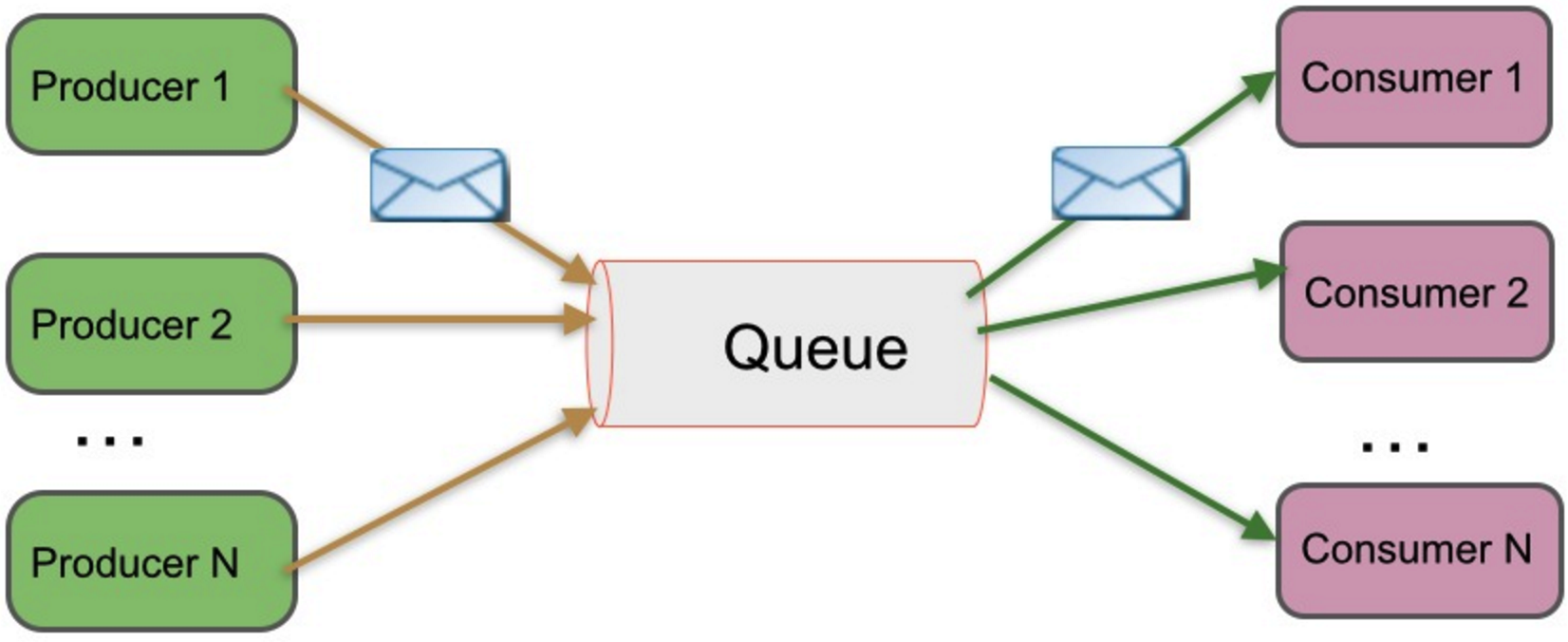
## 消息系统分类

Peer-to-Peer

Ø 一般基于Pull或者Polling接收消息

Ø 发送到队列中的消息被一个而且仅仅一个接收者所接收，即使有多个接收者在同一个队列中侦听同一消息

Ø 即支持异步“即发即弃”的消息传送方式，也支持同步请求/应答传送方式

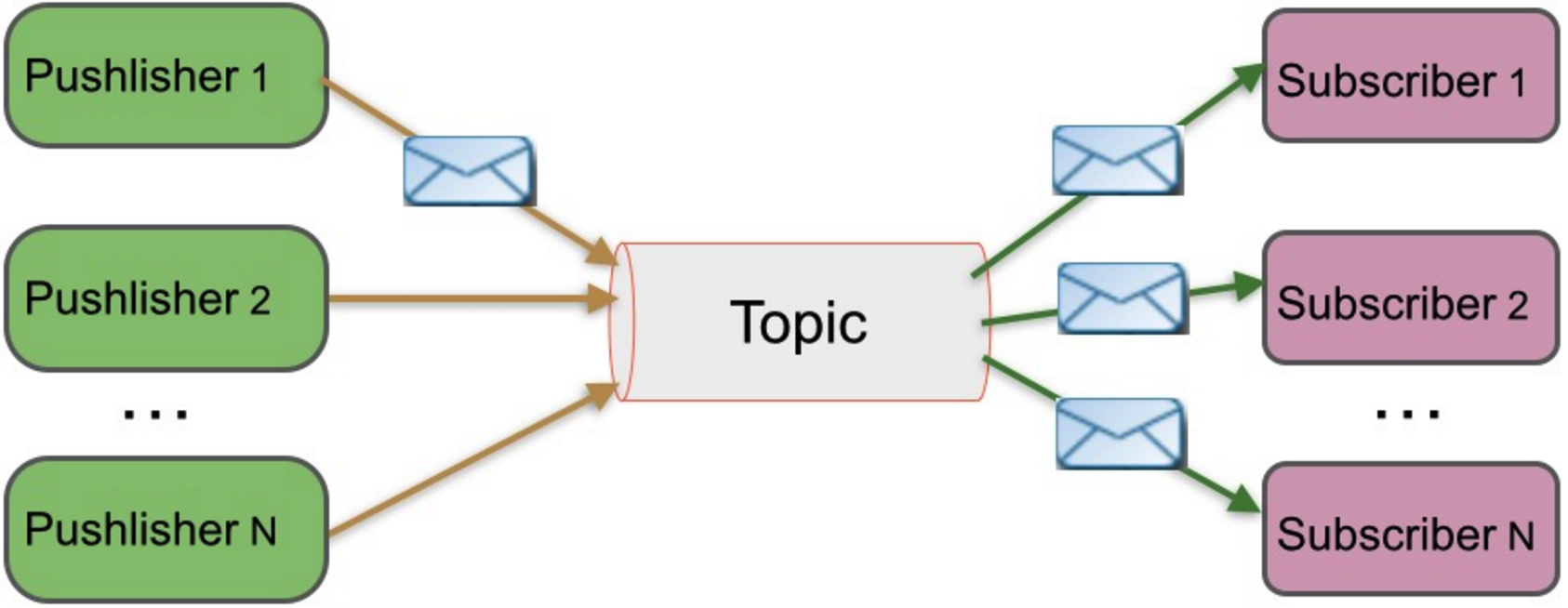


发布/订阅

Ø 发布到一个主题的消息，可被多个订阅者所接收

Ø 发布/订阅即可基于Push消费数据，也可基于Pull或者Polling消费数据

Ø 解耦能力比P2P模型更强



## 消息系统适用场景

解耦 各位系统之间通过消息系统这个统一的接口交换数据，无须了解彼此的存在

冗余 部分消息系统具有消息持久化能力，可规避消息处理前丢失的风险

扩展 消息系统是统一的数据接口，各系统可独立扩展

峰值处理能力 消息系统可顶住峰值流量，业务系统可根据处理能力从消息系统中获取并处理对应量的请求

可恢复性 系统中部分组件失效并不会影响整个系统，它恢复后仍然可从消息系统中获取并处理数据

异步通信 在不需要立即处理请求的场景下，可以将请求放入消息系统，合适的时候再处理

## 常用消息系统对比

RabbitMQ Erlang编写，支持多协议 AMQP，XMPP，SMTP，STOMP。支持负载均衡、数据持久化。同时支持Peer-to-Peer和发布/订阅模式

Redis 基于Key-Value对的NoSQL数据库，同时支持MQ功能，可做轻量级队列服务使用。就入队操作而言，Redis对短消息（小于10KB）的性能比RabbitMQ好，长消息的性能比RabbitMQ差。

ZeroMQ 轻量级，不需要单独的消息服务器或中间件，应用程序本身扮演该角色，Peer-to-Peer。它实质上是一个库，需要开发人员自己组合多种技术，使用复杂度高

ActiveMQ JMS实现，Peer-to-Peer，支持持久化、XA事务

Kafka/Jafka 高性能跨语言的分布式发布/订阅消息系统，数据持久化，全分布式，同时支持在线和离线处理

MetaQ/RocketMQ 纯Java实现，发布/订阅消息系统，支持本地事务和XA分布式事务

## Kafka设计目标

高吞吐率 在廉价的商用机器上单机可支持每秒100万条消息的读写

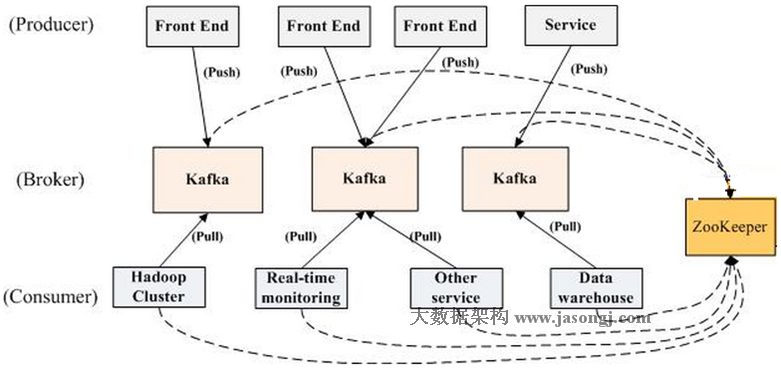
消息持久化 所有消息均被持久化到磁盘，无消息丢失，支持消息重放

完全分布式 Producer，Broker，Consumer均支持水平扩展

同时适应在线流处理和离线批处理

# Kafka架构

## Kafka架构简介



## 如何安装和使用Kafka集群

安装Docker

创建CentOS容器

安装JDK 1.7+

安装Zookeeper

启动Kafka broker

## Topic & Partition

Topic

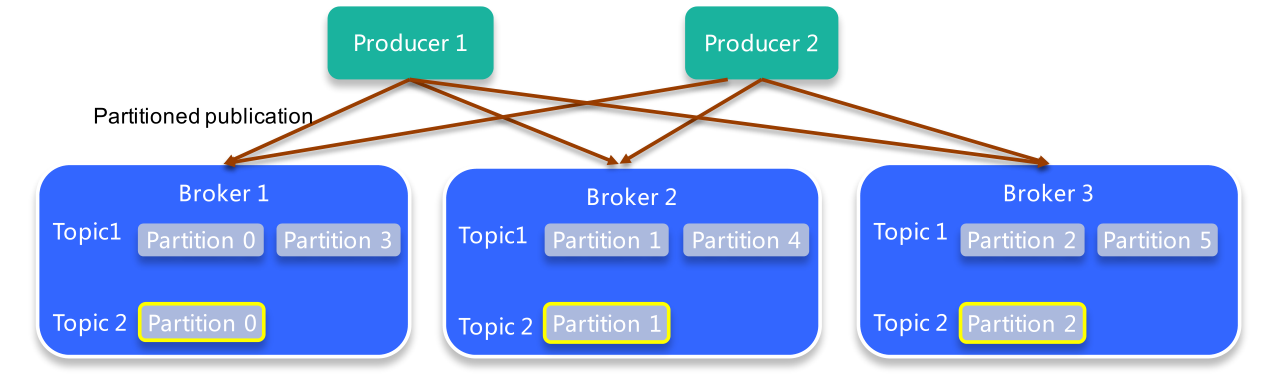
Ø 逻辑概念，同一个Topic的消息可分布在一个或多个节点（Broker）上

Ø 一个Topic包含一个或者多个Partition

Ø 每条消息都属于且仅属于一个Topic

Ø Producer发布数据时，必须指定将该消息发布到哪一个Topic

Ø Consumer订阅消息时，也必须指定订阅哪个Topic的消息



## Partition

Ø 物理概念，一个Partition只分布于一个Broker上（不考虑备份）

Ø 一个Partition物理上对应一个文件夹

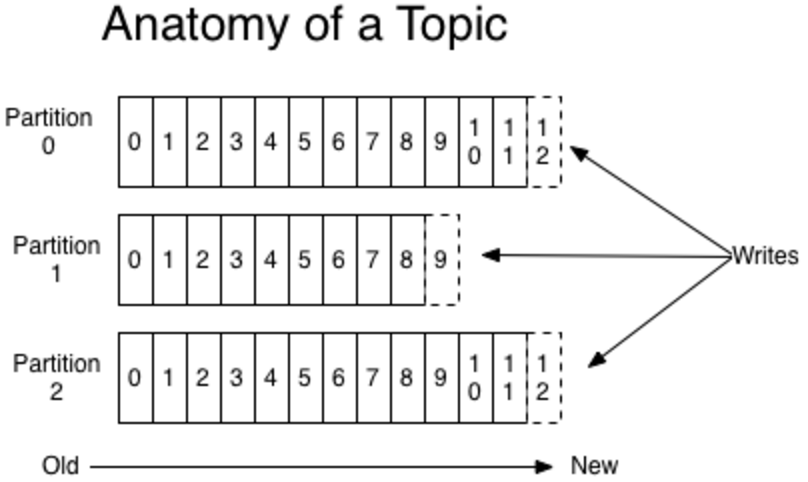
Ø 一个Partition包含多个Segment（Segment对用户透明）

Ø 一个Segment对应一个文件

Ø Segment由一个个不可变记录组成

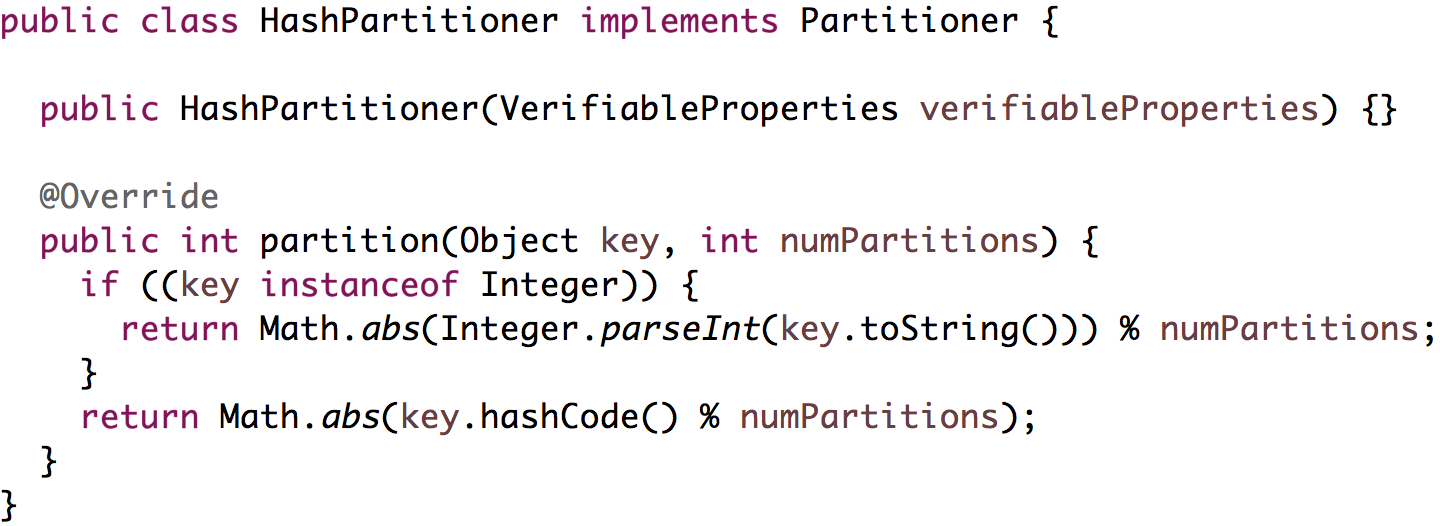
Ø 记录只会被append到Segment中，不会被单独删除或者修改

Ø 清除过期日志时，直接删除一个或多个Segment

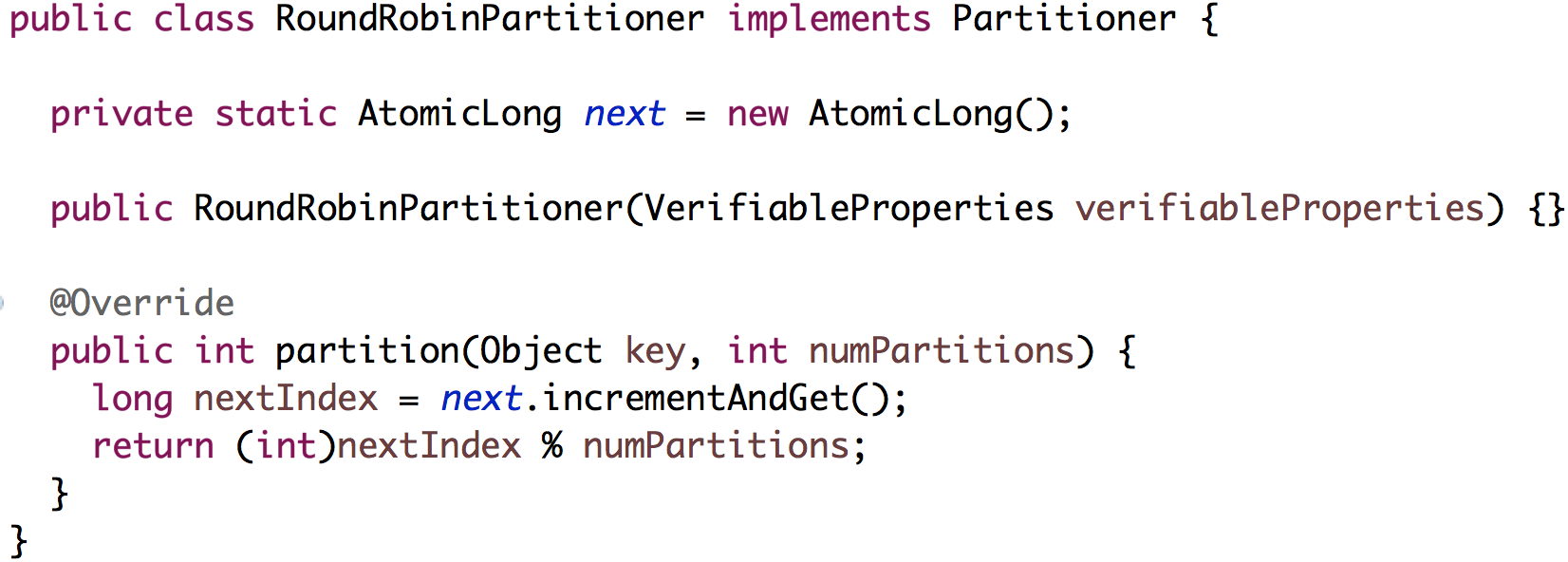


## Partitioner

Sample Hash Partitioner



Sample Round Robin Partitioner



## Sync Producer vs. Async Producer

Sync Producer

Ø 低延迟

Ø 低吞吐率

Ø 无数据丢失

Aync Producer

Ø 高延迟

Ø 高吞吐率

Ø 可能会有数据丢失

# Kafka数据复制与Failover

## CAP理论

Consistency

Ø 通过某个节点的写操作结果对后面通过其它节点的读操作可见

Ø 如果更新数据后，并发访问情况下可立即感知该更新，称为强一致性

Ø 如果允许之后部分或者全部感知不到该更新，称为弱一致性

Ø 若在之后的一段时间（通常该时间不固定）后，一定可以感知该更新，称为最终一致性

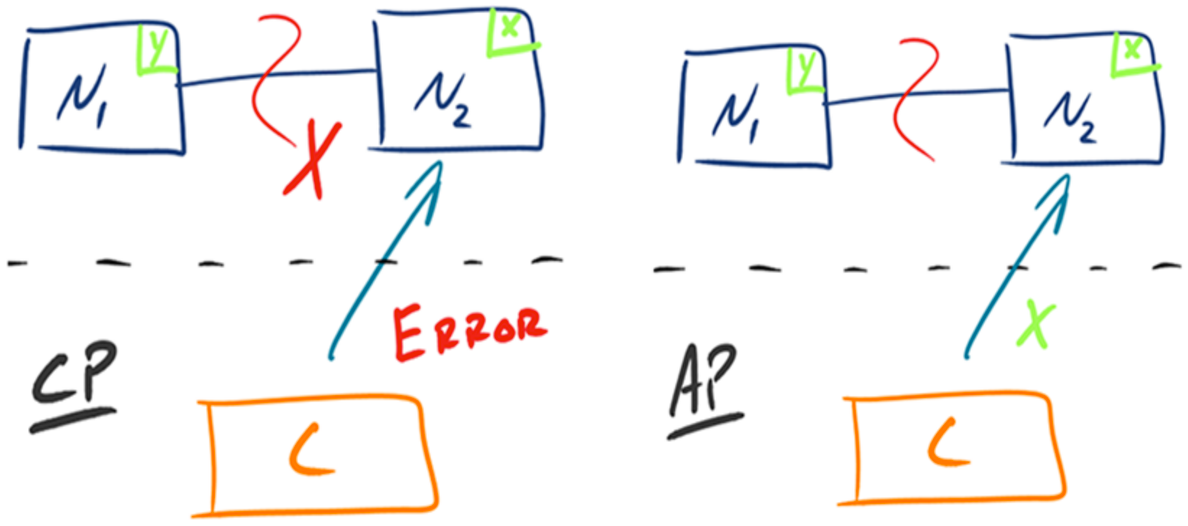
Availability

Ø 任何一个没有发生故障的节点必须在有限的时间内返回合理的结果

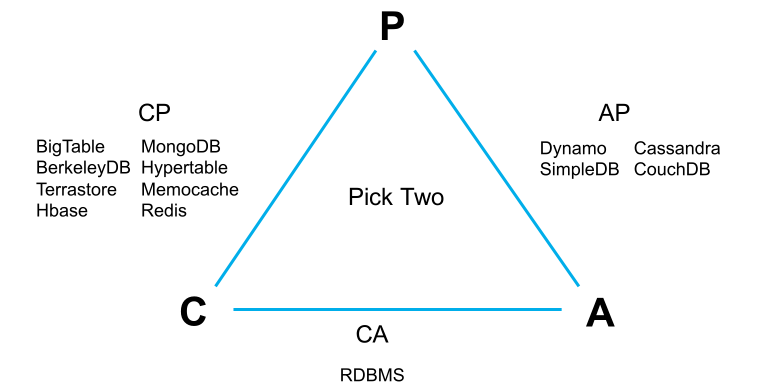
Partition tolerance

Ø 部分节点宕机或者无法与其它节点通信时，各分区间还可保持分布式系统的功能

CAP理论：分布式系统中，一致性、可用性、分区容忍性最多只可同时满足两个



一般分区容忍性都要求有保障，因此很多时候是在可用性与一致性之间做权衡



## 一致性方案

Master-slave

Ø RDBMS的读写分离即为典型的Master-slave方案

Ø 同步复制可保证强一致性但会影响可用性

Ø 异步复制可提供高可用性但会降低一致性

WNR

Ø 主要用于去中心化（P2P）的分布式系统中。DynamoDB与Cassandra即采用此方案

Ø N代表副本数，W代表每次写操作要保证的最少写成功的副本数，R代表每次读至少读取的副本数

Ø 当W+R>N时，可保证每次读取的数据至少有一个副本具有最新的更新

Ø 多个写操作的顺序难以保证，可能导致多副本间的写操作顺序不一致，Dynamo通过向量时钟保证最终一致性

Paxos及其变种

Ø Google的Chubby，Zookeeper的Zab，RAFT等

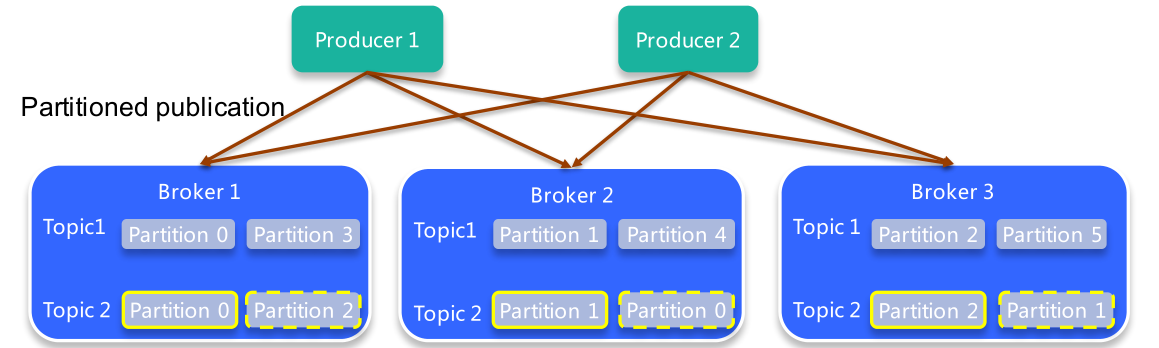
## Replica

Replica

Ø 当某个Topic的replication-factor为N且N大于1时，每个Partition都会有N个副本（Replica）

Ø Replica的个数小于等于Broker数，即对每个Partition而言每个Broker上只会有一个Replica，因此可用Broker ID表示Replica

Ø 所有Partition的所有Replica默认情况会均匀分布到所有Broker上



## Data Replication要解决的问题

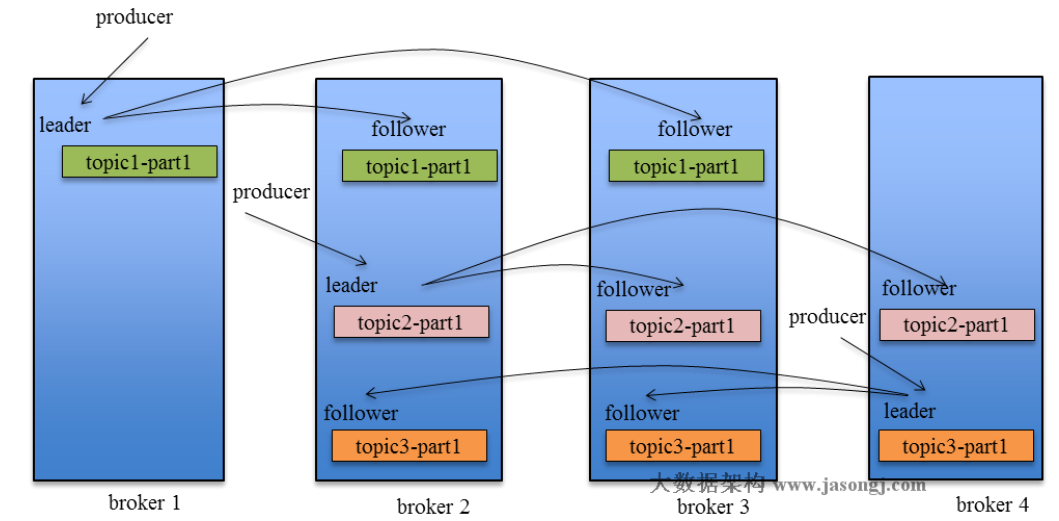
如何Propagate消息

何时Commit

如何处理Replica恢复

如何处理Replica全部宕机

## 如何Propagate消息



## 何时Commit

ISR

Ø 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）

Ø 如果一个Follower比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

Ø 当ISR中所有Replica都向Leader发送ACK时，Leader即Commit

Commit策略

Ø Server配置

o replica.lag.time.max.ms=10000

o replica.lag.max.messages=4000

Ø Topic配置

o min.insync.replicas=1

Ø Producer配置

o request.required.acks=0

## 如何处理Replica恢复

1. ISR={A,B,C}

Leader A commit m1

2. A fails, B is new leader

ISR={B,C}

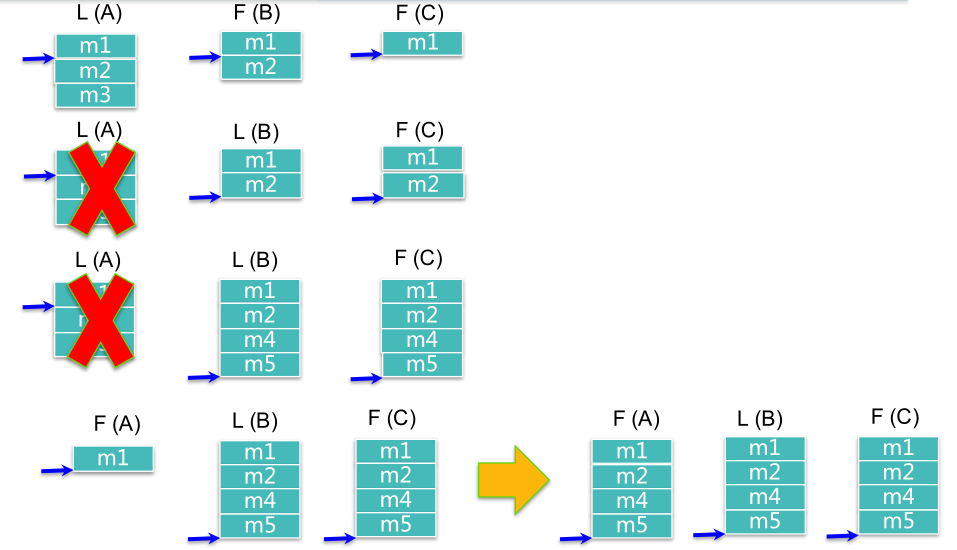
B commit m2, not m3

3. B commit m4, m5

4. A comes back, truncates

to m1 and catch up

finally ISR={A,B,C}



## 如何处理Replica全部宕机

等待ISR中任一Replica恢复，并选它为Leader

Ø 等待时间较长，降低可用性

Ø 或ISR中的所有Replica都无法恢复或者数据丢失，则该Partition将永不可用

选择第一个恢复的Replica为新的Leader，无论它是否在ISR中

Ø 并未包含所有已被之前Leader Commit过的消息，因此会造成数据丢失

Ø 可用性较高

# Kafka如何使用Zookeeper

## Zookeeper是什么

Zookeeper是一个高性能分布式应用协调服务

Ø Naming Service

Ø 配置管理

Ø Leader Election

Ø 服务发现

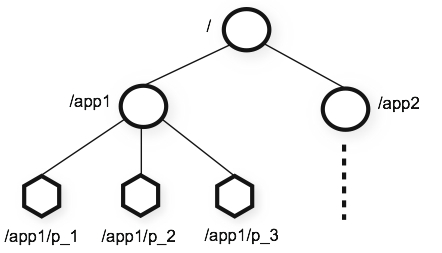
Ø 同步

Ø Group Service

Ø Barrier

Ø 分布式队列

Ø 两阶段提交



## Zookeeper工作方式

Ø Zookeeper集群包含1个Leader，多个Follower。

Ø 所有的Follower都可提供读服务

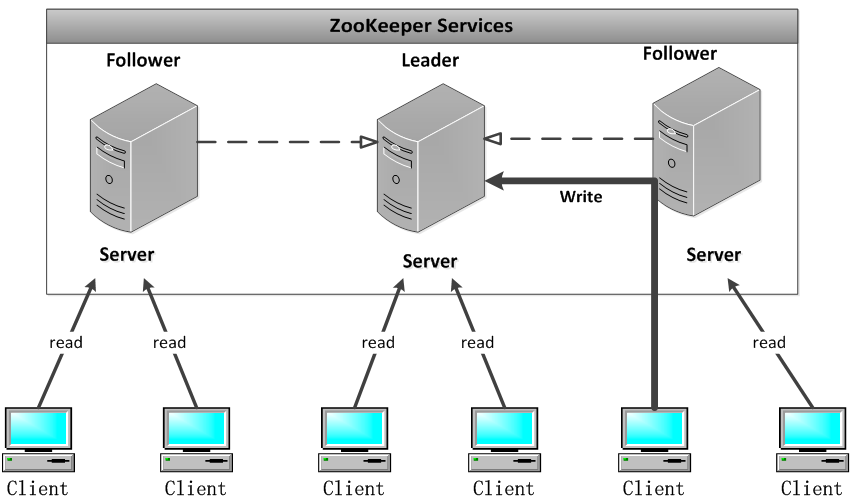
Ø 所有的写操作都会被forward到Leader

Ø Client与Server通过NIO通信。

Ø 全局串行化所有的写操作

Ø 保证同一客户端的指令被FIFO执行

Ø 保证消息通知的FIFO



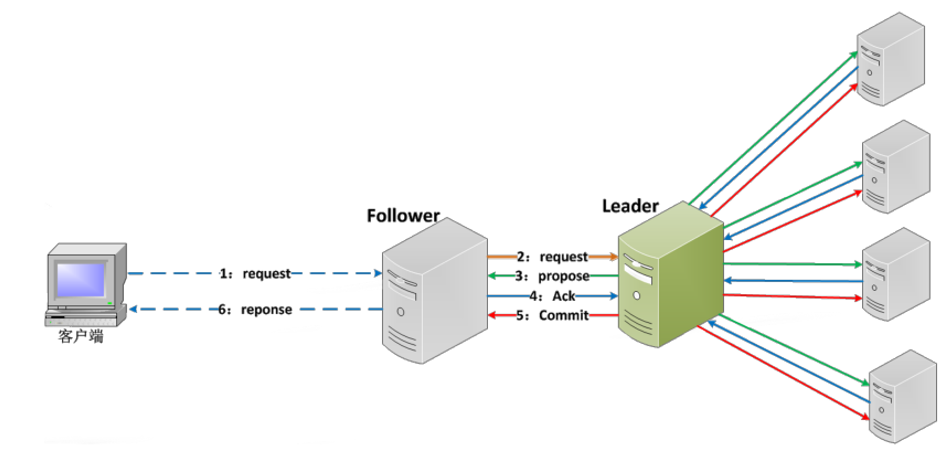
## Zab协议 – 广播模式

Ø Leader将所有更新（称为proposal），顺序发送给Follower

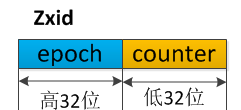
Ø 当Leader收到半数以上的Follower对此proposal的ACK时，即向所有Follower发送commit消息，并在本地commit该消息

Ø Follower收到Proposal后即将该Proposal写入磁盘，写入成功即返回ACK给Leader

Ø 每个Proposal都有一个唯一的单调递增的proposal ID，即zxid



## Zab协议 – 恢复模式



进入恢复模式 当Leader宕机或者丢失大多数Follower后，即进入恢复模式

结束恢复模式 新领导被选举出来，且大多数Follower完成了与Leader的状态同步后，恢复模式即结束，从而进入广播模式

恢复模式的意义

Ø 发现集群中被commit的proposal的最大zxid

Ø 建立新的epoch，从而保证之前的Leader不能再commit新的Proposal

Ø 集群中大部分节点都commit过前一个Leader commit过的消息，而新的Leader是被大部分节点所支持的，所以被之前Leader commit过的Proposal不会丢失，至少被一个节点所保存

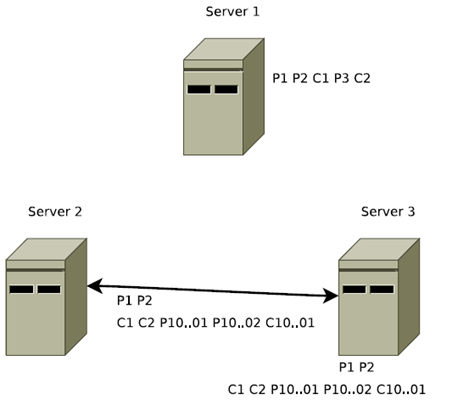
Ø 新Leader会与所有Follower通信，从而保证大部分节点都拥有最新的数据

## Zab协议 – 恢复模式

恢复阶段的保证

Ø 若一条消息在一台机器上被deliver，那么该消息必须将在每台机器上deliver，即使那台机器故障了

Ø 一个被skip的消息，必须仍然需要被skip



## Zookeeper一致性保证

顺序一致性 从一个客户端发出的更新操作会按发送顺序被顺序执行

原子性 更新操作要么成功要么失败，无中间状态

单一系统镜像 一个客户端只会看到同一个view，无论它连到哪台服务器

可靠性

Ø 一旦一个更新被应用，该更新将被持久化，直到有客户端更新该结果

Ø 如果一个客户端得到更新成功的状态码，则该更新一定已经生效

Ø 任何一个被客户端通过读或者更新“看到”的结果，将不会被回滚，即使是从失败中恢复

实时性 保证客户端可在一定时间（通常是几十秒）内看到最新的视图

## Zookeeper使用注意事项

Ø 只保证同一客户端的单一系统镜像，并不保证多个不同客户端在同一时刻一定看到同一系统镜像，如果要实现这种效果，需要在读取数据之前调用sync操作

Ø Zookeeper读性能好于写性能，因为任何Server均可提供读服务，而只有Leader可提供写服务

Ø 为了保证Zookeeper本身的Leader Election顺利进行，通常将Server设置为奇数

Ø 若需容忍f个Server的失败，必须保证有2f+1个以上的Server

## Kafka如何使用Zookeeper

Ø 配置管理

Ø Leader Election

Ø 服务发现

# Kafka Leader Election

## Zookeeper的基本操作

四种节点类型 PERSIST, PERSIST\_SEQUENTIAL, EPHEMERAL, EPHEMERAL\_SEQUENTIAL

可注册Watch操作

Ø Created event: Enabled with a call to exists

Ø Deleted event: Enabled with a call to exists, getData, and getChildren

Ø Changed event: Enabled with a call to exists and getData

Ø Child event: Enabled with a call to getChildren

Watch特征

Ø 客户端先得到通知再得到数据

Ø Watch被fire后即取消，不会再Watch后续变化

## 基于Zookeeper的Leader Election

抢注Leader节点——非公平模式

1. 创建Leader父节点，如 /chroot，并将其设置为persist节点

2. 各客户端通过在/chroot下创建Leader节点，如/chroot/leader，来竞争Leader。该节点应被设置为ephemeral

3. 若某创建Leader节点成功，则该客户端成功竞选为Leader

4. 若创建Leader节点失败，则竞选Leader失败，在/chroot/leader节点上注册exist的watch，一旦该节点被删除则获得通知

5. Leader可通过删除Leader节点来放弃Leader

6. 如果Leader宕机，由于Leader节点被设置为ephemeral，Leader节点会自行删除。而其它节点由于在Leader节点上注册了watch，故可得到通知，参与下一轮竞选，从而保证总有客户端以Leader角色工作

先到先得，后者监视前者——公平模式

1. 创建Leader父节点，如 /chroot，并将其设置为persist节点

2. 各客户端通过在/chroot下创建Leader节点，如/chroot/leader，来竞争Leader。该节点应被设置为ephemeral\_sequential

3. 客户端通过getChildren方法获取/chroot/下所有子节点，如果其注册的节点的id在所有子节点中最小，则当前客户端竞选Leader成功

4. 否则，在前面一个节点上注册watch，一旦前者被删除，则它得到通知，返回step 3（并不能直接认为自己成为新Leader，因为可能前面的节点只是宕机了）

5. Leader节点可通过自行删除自己创建的节点以放弃Leader

## Leader Election在Curator中的实现

LeaderLatch

Ø 竞选为Leader后，不可自行放弃领导权

Ø 只能通过close方法放弃领导权

Ø 强烈建议增加ConnectionStateListener，当连接SUSPENDED或者LOST时视为丢失领导权

Ø 可通过await方法等待成功获取领导权，并可加入timeout

Ø 可通过hasLeadership方法判断是否为Leader

Ø 可通过getLeader方法获取当前Leader

Ø 可通过getParticipants方法获取当前竞选Leader的参与方

LeaderSelector

Ø 竞选Leader成功后回调takeLeadership方法

Ø 可在takeLeadership方法中实现业务逻辑

Ø 一旦takeLeadership方法返回，即视为放弃领导权

Ø 可通过autoRequeue方法循环获取领导权

Ø 可通过hasLeadership方法判断是否为Leader

Ø 可通过getLeader方法获取当前Leader

Ø 可通过getParticipants方法获取当前竞选Leader的参与方

## Kafka“各自为政”Leader Election

“各自为政”Leader Election

Ø 每个Partition的多个Replica同时竞争Leader

优点

Ø 实现简单

缺点

Ø Herd Effect

Ø Zookeeper负载过重

Ø Latency较大

## Kafka基于Controller的Leader Election

基于Controller的Leader Election

Ø 整个集群中选举出一个Broker作为Controller

Ø Controller为所有Topic的所有Partition指定Leader及Follower

优点

Ø 极大缓解Herd Effect问题

Ø 减轻Zookeeper负载

Ø Controller与Leader及Follower间通过RPC通信，高效且实时

缺点

Ø 引入Controller增加了复杂度

Ø 需要考虑Controller的Failover

# Kafka Consumer

## Pull vs. Push

Producer Producer通过主动Push的方式将消息发布到Broker

Consumer Consumer通过Pull从Broker消费数据

Ø Push

o 优势：延时低

o 劣势：可能造成Consumer来不及处理消息；网络拥塞

Ø Pull

o 优势：Consumer按实际处理能力获取相应量的数据；Broker实现简单

o 劣势：如果处理不好，实时性相对不足（Kafka使用long polling）

## High Level Consumer

Ø 很多应用场景下，客户程序只是希望从Kafka顺序读取并处理数据，而不太关心具体的offset。

Ø 同时也希望提供一些语义，例如同一条消息只被某一个Consumer消费（单播）或被所有Consumer消费（广播）

Ø Kafka High Level API提供了一个从Kafka消费数据的高层抽象，从而屏蔽掉其中的细节，并提供丰富的语义。

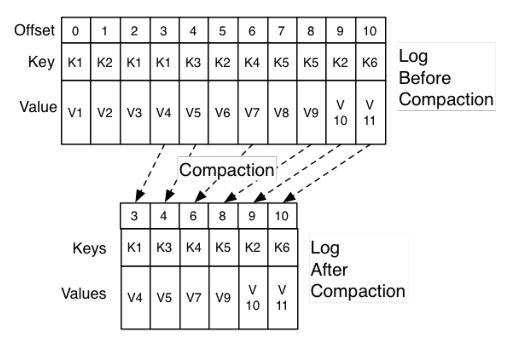
## Consumer Group

Ø High Level Consumer将从某个Partition读取的最后一条消息的offset存于Zookeeper中（从0.8.2开始同时支持将offset存于Zookeeper中和专用的Kafka Topic中）。

Ø 这个offset基于客户程序提供给Kafka的名字来保存，这个名字被称为Consumer Group。

Ø Consumer Group是整个Kafka集群全局唯一的，而非针对某个Topic的。

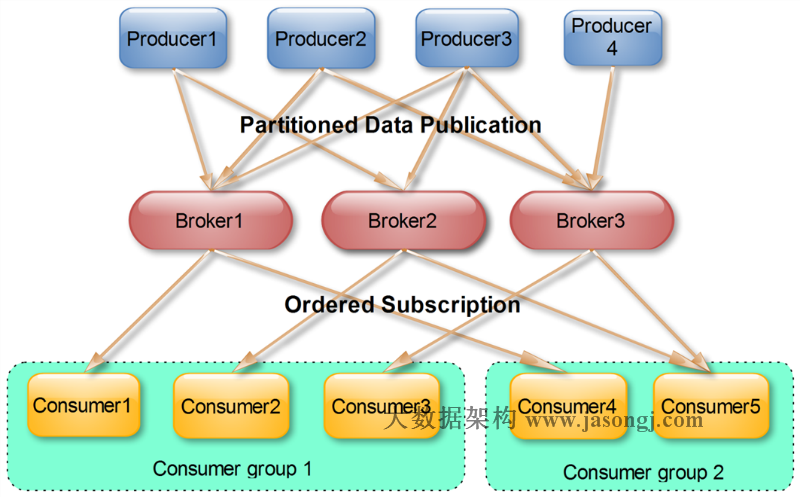
Ø 每个High Level Consumer实例都属于一个ConsumerGroup，若不指定则属于默认的Group



Ø 消息被消费后，并不会被删除，只是相应的offset加一

Ø 对于每条消息，在同一个Consumer Group里只会被一个Consumer消费

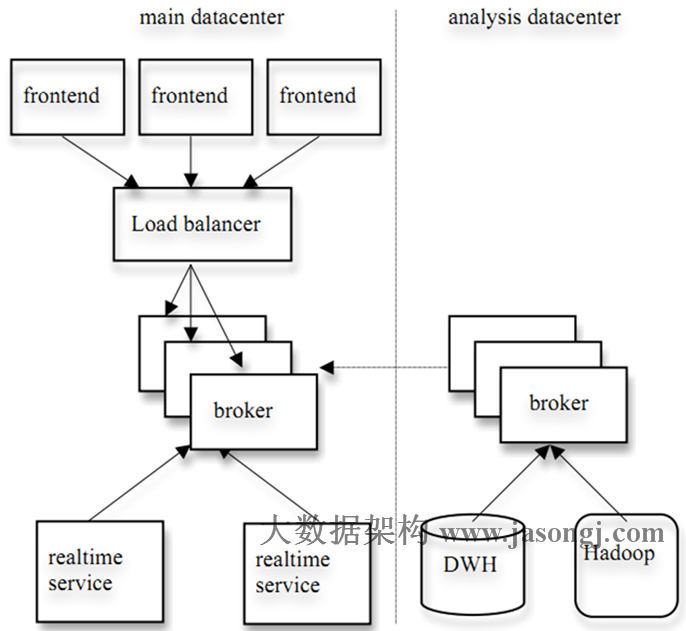
Ø 不同Consumer Group可消费同一条消息



Ø Kafka的设计理念之一就是同时提供对离线批处理和在线流处理的支持。

Ø 可同时使用Hadoop系统进行离线批处理，Storm或其它流处理系统进行流处理。

Ø 可使用Kafka的Mirror Maker将消息从一个数据中心镜像到另一个数据中心。



## High Level Consumer Rebalance

Consumer启动及Rebalance流程

Ø High Level Consumer启动时将其ID注册到其Consumer Group下，在Zookeeper上的路径为/consumers/[consumer group]/ids/[consumer id]

Ø 在/consumers/[consumer group]/ids上注册Watch

Ø 在/brokers/ids上注册Watch

Ø 如果Consumer通过Topic Filter创建消息流，则它会同时在/brokers/topics上也创建Watch

Ø 强制自己在其Consumer Group内启动Rebalance流程

## High Level Consumer Rebalance

Consumer Rebalance算法

Ø 将目标Topic下的所有Partirtion排序，存于P T

Ø 对某Consumer Group下所有Consumer排序，存于C G ，第i个Consumer记为C i

Ø N=size(P T )/size(C G ) ，向上取整

Ø 解除C i 对原来分配的Partition的消费权（i从0开始）

Ø 将第 i∗N 到（i+1）∗N−1个Partition分配给C

## High Level Consumer Rebalance

Consumer Rebalance算法缺陷及改进

Ø Herd Effect 任何Broker或者Consumer的增减都会触发所有的Consumer的Rebalance

Ø Split Brain 每个Consumer分别单独通过Zookeeper判断哪些Broker和Consumer宕机，同时Consumer在同一时刻从Zookeeper“看”到的View可能不完全一样，这是由Zookeeper的特性决定的。

Ø 调整结果不可控 所有Consumer分别进行Rebalance，彼此不知道对应的Rebalance是否成功

## Low Level Consumer

使用Low Level Consumer (Simple Consumer)的主要原因是，用户希望比Consumer Group更好的控制数据的消费，如

Ø 同一条消息读多次，方便Replay

Ø 只消费某个Topic的部分Partition

Ø 管理事务，从而确保每条消息被处理一次（Exactly once）与High Level Consumer相对，Low Level Consumer要求用户做大量的额外工作

Ø 在应用程序中跟踪处理offset，并决定下一条消费哪条消息

Ø 获知每个Partition的Leader

Ø 处理Leader的变化

Ø 处理多Consumer的协作

# Kafka Consumer Offset Management

## Simple Consumer

手工管理offset

Ø 每次从特定Partition的特定offset开始fetch特定大小的消息

Ø 完全由Consumer应用程序决定下一次fetch的起始offset

使用Low Level Consumer (Simple Consumer)的主要原因是，用户希望比Consumer Group更好的控制数据的消费，如

Ø 同一条消息读多次，方便Replay

Ø 只消费某个Topic的部分Partition

Ø 管理事务，从而确保每条消息被处理一次（Exactly once）与High Level Consumer相对，Low Level Consumer要求用户做大量的额外工作

Ø 在应用程序中跟踪处理offset，并决定下一条消费哪条消息

Ø 获知每个Partition的Leader

Ø 处理Leader的变化

Ø 处理多Consumer的协作

## High Level Consumer

自动管理offset

Ø auto.commit.enable=true

Ø auto.commit.interval.ms=60 \* 1000

手工管理offset

Ø ConsumerConnector.commitOffsets();

offset存储

Ø offsets.storage=zookeeper

Ø dual.commit.enabled=true

## Log compaction

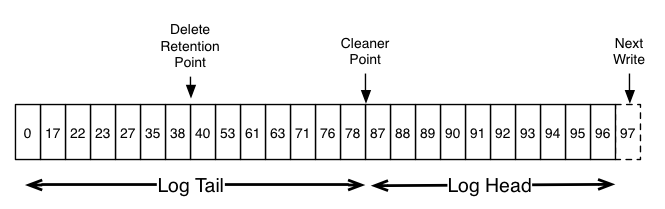
Log compaction效果

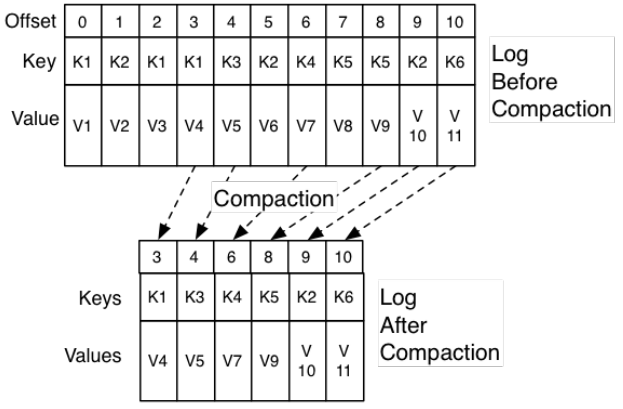
Ø 一直保持消费Log head的consumer可按顺序消费所有消息，并且offset连续

Ø 任何从offset 0开始的读操作至少可读到每个key对应的最后一条消息

Ø 每条消息的offset保持不变，offset是消息的永久标志符

Ø 消费本身的顺序不会被改变





## New API design

## New API

Producer

Ø 增加发送回调

Ø 重构Partitioner接口

统一High Level API与Low Level API

Ø 从kafka.consumer和kafka.javaapi到kafka.clients.consumer

Ø subscribe动态rebalance vs. assign手动分配

Ø 将offset存于kafka和zookeeper以外

Ø ConsumerRebalanceListener

Ø 控制消费位置

Ø 消费流程控制

## Producer

Producer

Ø 增加发送回调

Ø 重构Partitioner接口

## 统一包名

从kafka.consumer和kafka.javaapi到kafka.clients.consumer

## subscribe vs. assign

Subscribe

Ø 自动Rebalance

Ø 自动分配Partition给Consumer

Ø 使用subscribe接口，并可注册ConsumerRebalanceListener

Assign

Ø 使用Assign接口

Ø 指定目标Partition

## offset管理

自动commit

Ø 自动commit

手动commit

Ø 手动commit全部offset

Ø 手动commit特定partition的offset

Ø 支持同步和异步commit并支持commit回调

## 消费流程控制

可暂停/恢复对某些Partition的消费

Ø pause 暂停消费指定的Partition

Ø resume 恢复对指定partition的消费

Ø wakeup 唤醒poll阻塞，并抛出WakeupException

# Kafka Stream

## Agenda

Overview

Concepts

Ø Topology

Ø Time

Ø States

Window

Ø Hopping time window

Ø Tumbling time windows

Ø Sliding windows

Join

API

Ø Low-Level Processor API

Ø High-Level DSL API

## Overview

Kafka Stream特点

Ø 简单轻量的SDK

Ø 除了对Kafka本身的依赖外，无外部依赖

Ø 支持容错的local state从而支持高效的状态操作，如Join和Window操作

Ø Record级别的处理

Ø 提供两种处理原语，Processor API和DSL

## Concepts

Stream Topology

Ø Stream 时间上无界的，有序的，不可变数据集

Ø Stream processing application 通过一个或多个Topology定义的计算逻辑

Ø Stream processor 一个计算原语，类似于Storm的Bolt

Time

Ø Event Time 消息创建时间，一般由消费携带

Ø Processing Time 消息被处理时的时间

Ø Ingestion Time 消息存入Topic/Partition时的时间

State

Ø In-memory State Store

Ø Persistent State Store

## Window

Hopping time windows

Ø Advance interval 结果输出interval

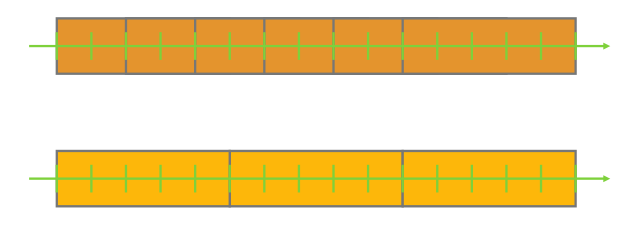
Ø Window size 计算数据集

Tumbling time windows

Ø Hopping time windows的特例

Sliding windows

Ø 只用于Join操作，可由JoinWindow类指定



## KStream vs. KTable

KStream

Ø KStream为数据流，每条消息代表一条不可变的新记录

KTable

Ø KTable为change log流，每条消息代表一个更新，几条key相同的消息会将该key的值更新为最后一条消息的值

Example

Ø 对于KStream和KTable中插入两条消息 (“key1”, 1), (“key1”, 2)

Ø 对KStream作sum，结果为(“key1”,3)

Ø 对KTable作sum，结果为(“key1”,2)

## Join

KStream-KStream Join

Ø 适用于Window Join

Ø 结果为KStream

KStream-KTable Join

Ø KTable的变化只影响KStream中新数据

Ø 新结果的输出由KStream驱动

Ø 输出为KStream

KTable-KTable Join

Ø 类似于RDBMS的Join

Ø 结果为KTable

## Stream API

Low-level processor API

High-level DSL API

# Kafka高性能之道

高效使用磁盘

零拷贝

批处理和压缩

Partition

ISR

## 高效使用磁盘

顺序写磁盘 顺序写磁盘性能高于随机写内存

Append Only 数据不更新，无记录级的数据删除（只会整个segment删除）

充分利用Page Cache

Ø I/O Scheduler将连续的小块写组装成大块的物理写从而提高性能

Ø I/O Scheduler会尝试将一些写操作重新按顺序排好，从而减少磁盘头的移动时间

Ø 充分利用所有空闲内存（非JVM内存）

o 应用层cache也会有对应的page cache与之对应，直接使用page cache可增大可用cache

o 如使用heap内的cache，会增加GC负担

Ø 读操作可直接在page cache内进行。如果进程重启，JVM内的cache会失效，但page cache仍然可用

Ø 可通过如下参数强制flush，但并不建议这么做

o log.flush.interval.messages=10000

o log.flush.interval.ms=1000

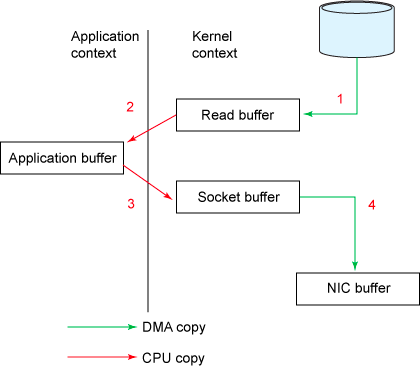
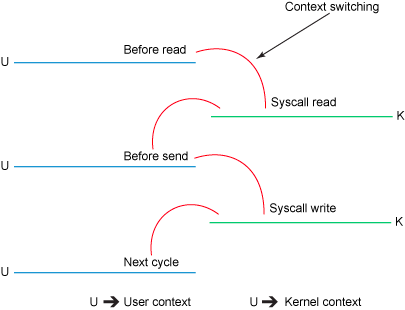
支持多Directory（可使用多Drive）

## 零拷贝

传统模式下数据从文件传输到网络需要4次数据拷贝，4次上下文切换和2次系统调用

File.read(fileDesc, buf, len);

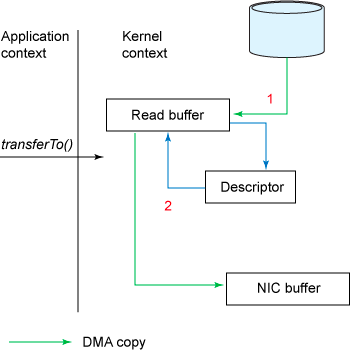
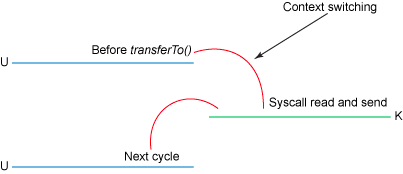
Socket.send(socket, buf, len);

## 零拷贝

通过NIO的transferTo/transferFrom调用操作系统的sendfile实现零拷贝。总共发生2次内核数据拷贝，2次上下文切换和1次系统调用，消除了CPU数据拷贝

public void transferTo(long position, long count, WritableByteChannel target);

## 批处理和压缩

Producer和Consumer均支持批量处理数据，从而减少了网络传输的开销

Producer可将数据压缩后发送给broker，从而减少网络传输代价。目前支持Snappy, Gzip和LZ4压缩

## Partition

通过Partition实现了并行处理和水平扩展

Partition是Kafka（包括Kafka Stream）并行处理的最小单位

不同Partition可处于不同的Broker（节点），充分利用多机资源

同一Broker（节点）上的不同Partition可置于不同的Directory，如果节点上有多个Disk Drive，可将不同的Drive对应不同的Directory，从而使Kafka充分利用多Disk Drive的磁盘优势

## ISR

ISR实现了可用性和一致性的动态平衡

Ø replica.lag.time.max.ms=10000

Ø replica.lag.max.messages=4000

ISR可容忍更多的节点失败

Ø Majority Quorum如果要容忍f个节点失败，则至少需要2f+1个节点

Ø ISR如果要容忍f个节点失败，至少需要f+1个节点

如何处理Replica Crash

Ø Leader crash后，ISR中的任何replica皆可竞选成为Leader

Ø 如果所有replica都crash，可选择让第一个recover的replica或者第一个在ISR中的replica成为leader

Ø unclean.leader.election.enable=true

# Kafka Security

## Kafka Security

Kafka Security

Ø SSL/TLS

Ø SASL (Kerberos)

## SSL/TLS原理

摘要算法 将需加密的明文使用hash函数生成一段被称为摘要的一串密文（128位）

数字签名 将摘要使用私钥加密后连同原明文一同发送，接收方使用公钥将其解密为摘要，然后使用hash函数从明文算出摘要，将二者进行对比，以确认 a)消息的完整性 b)消息确实是由发送方签名并发出

CA证书 用户将证书的公钥和个人信息发给CA，CA确信来源后，给用户发一个（由CA私钥）签名后的数字证书

Ø 证书颁发机构，证书本身的数字签名，证书持有者公钥，证书签名用到的hash算法

证书有效性

Ø 根据CA名找到相应的CA根证书和CA公钥，用CA公钥对证书摘要解密，若解密失败则认为其为假证书

Ø 若证书被篡改，用CA公钥对证书签名进行解密得到摘要A，再用hash函数计算出证书的摘要B，若A与B相同则认为证书正常，否则认为其被篡改

## Enable SSL

Enable SSL

Ø 生成服务器keystore （密钥和证书）

Ø 生成客户端keystore （密钥和证书）

Ø 创建CA证书

Ø 将CA证书导入服务器truststore

Ø 将CA证书导入客户端truststore

Ø 从服务器keystore导出证书

Ø 用CA证书给服务器证书签名

Ø 将CA证书导入服务器keystore

Ø 将CA证书导入客户端keystore

Ø 将已签名服务器证书导入服务器keystore

Ø 验证broker SSL是否生效

## Kafka Monitoring

Yammer Metrics

JMX

Kafka Managers

# Kafka运维

Topic增删改查

Partition/Replica重新分配

设置Preferred Replica Election

## Topic增删改查

Topic的设置，影响分布式处理的并行度，从而影响性能，甚至结果的正确性

bin/kafka-topics.sh

## Partition/Replica重新分配

Partition/Replica重新分配

Ø 增加Broker后，已有的数据不会自动迁移到新Broker上

Ø 删除Broker前，需要将待删除的Broker上的数据迁移到其它Broker上

bin/kafka-reassign-partitions.sh

## Partition/Replica重新分配

Partition/Replica重新分配过程

1. 将Zookeeper中的AR（Current Assigned Replicas）更新为OAR（Original list of replicas for partition） + RAR（Reassigned replicas）。

2. 强制更新Zookeeper中的leader epoch，向AR中的每个Replica发送LeaderAndIsrRequest

3. 将RAR - OAR中的Replica设置为NewReplica状态

4. 等待直到RAR中所有的Replica都与其Leader同步

5. 将RAR中所有的Replica都设置为OnlineReplica状态

6. 将Cache中的AR设置为RAR

7. 若Leader不在RAR中，则从RAR中重新选举出一个新的Leader并发送LeaderAndIsrRequest。若新的Leader不是从RAR中选举而出，则还要增加Zookeeper中的leader epoch

8. 将OAR - RAR中的所有Replica设置为OfflineReplica状态，该过程包含两部分。第一，将Zookeeper上ISR中的OAR - RAR移除并向Leader发送LeaderAndIsrRequest从而通知这些Replica已经从ISR中移除；第二，向OAR - RAR中的Replica发送StopReplicaRequest从而停止不再分配给该Partition的Replica

9. 将OAR - RAR中的所有Replica设置为NonExistentReplica状态从而将其从磁盘上删除

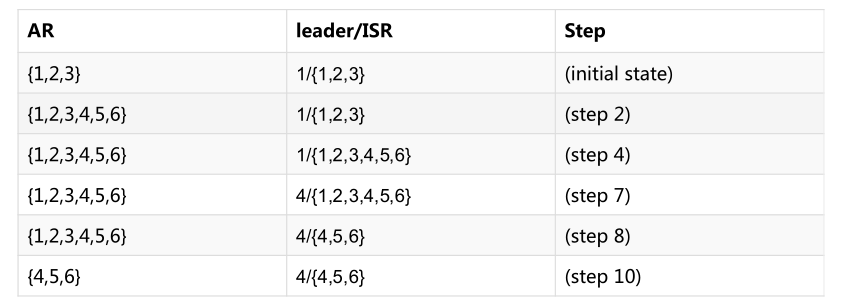
10. 将Zookeeper中的AR设置为RAR

11. 删除/admin/reassign\_partition

## Partition/Replica重新分配

Partition/Replica重新分配过程

以下是Partition重新分配的案例，OAR = ｛1，2，3｝，RAR = ｛4，5，6｝，Partition重新分配过程中Zookeeper中的AR和Leader/ISR路径如下



## Preferred Replica Leader Election

Preferred Replica Leader Election

Ø 所有的读写都发生在Leader上，Leader的分布是否均匀，直接影响集群的负载均衡

Ø Preferred Replica只有在ISR中时才会被选为Leader

bin/kafka-preferred-replica-election.sh

可通过如下配置来启用和控制自动Leadership平衡

Ø leader.imbalance.check.interval.seconds=300

Ø leader.imbalance.per.broker.percentage=10

Ø auto.leader.rebalance.enable=true

Preferred Replica Leader Election

1. 在Zookeeper上创建/admin/preferred\_replica\_election节点，并存入需要调整Preferred Replica的Partition信息

2. Controller一直Watch该节点，一旦该节点被创建，Controller会收到通知，并获取该内容

3. Controller读取Preferred Replica，如果发现该Replica当前并非是Leader并且它在该Partition的ISR中，Controller向该Replica发送LeaderAndIsrRequest，使该Replica成为Leader。如果该Replica当前并非是Leader，且不在ISR中，Controller为了保证没有数据丢失，并不会将其设置为Leader

# Kafka性能测试

## Kafka性能测试——Producer

8 vCPU

16 GB

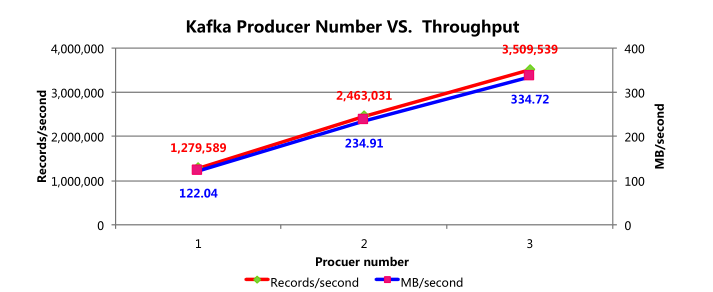
3 nodes / brokers

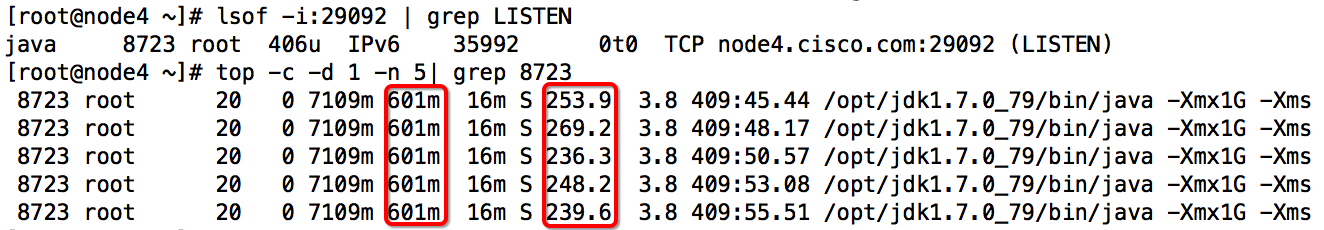
6 partitions

1 replica

async mode

message size = 100 bytes





## Kafka性能测试——Consumer

8 vCPU

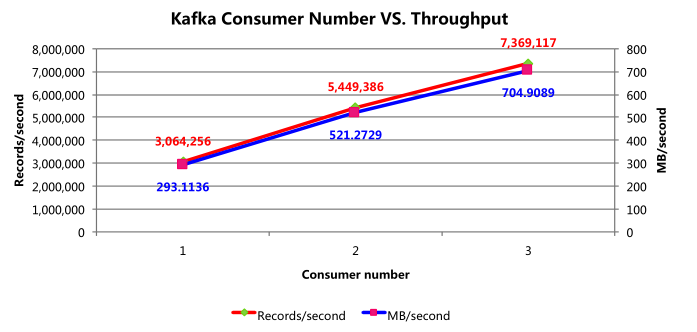
16 GB

3 nodes / brokers

6 partitions

1 replica

message size = 100 bytes



## Kafka性能测试——Partition

8 vCPU

16 GB

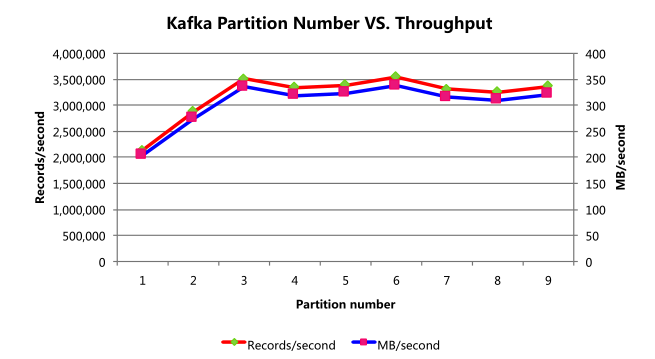
3 nodes / brokers

3 producers

1 replica

async mode

message size = 100 bytes



## Kafka性能测试——Message Size

8 vCPU

16 GB

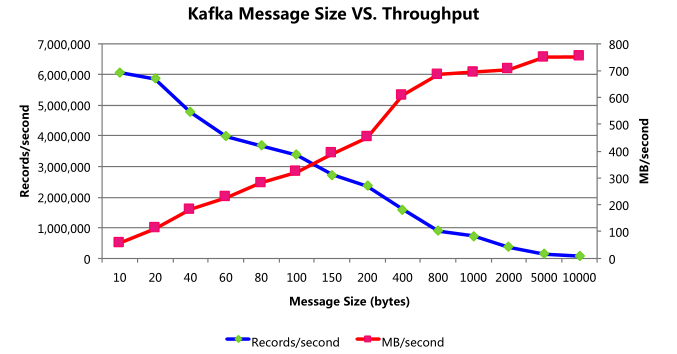
3 nodes / brokers

3 producers

6 partitions

1 replica

async mode



## Kafka性能测试——Replication

8 vCPU

16 GB

3 nodes / brokers

3 producers

6 partitions

async mode

message size = 100 bytes

