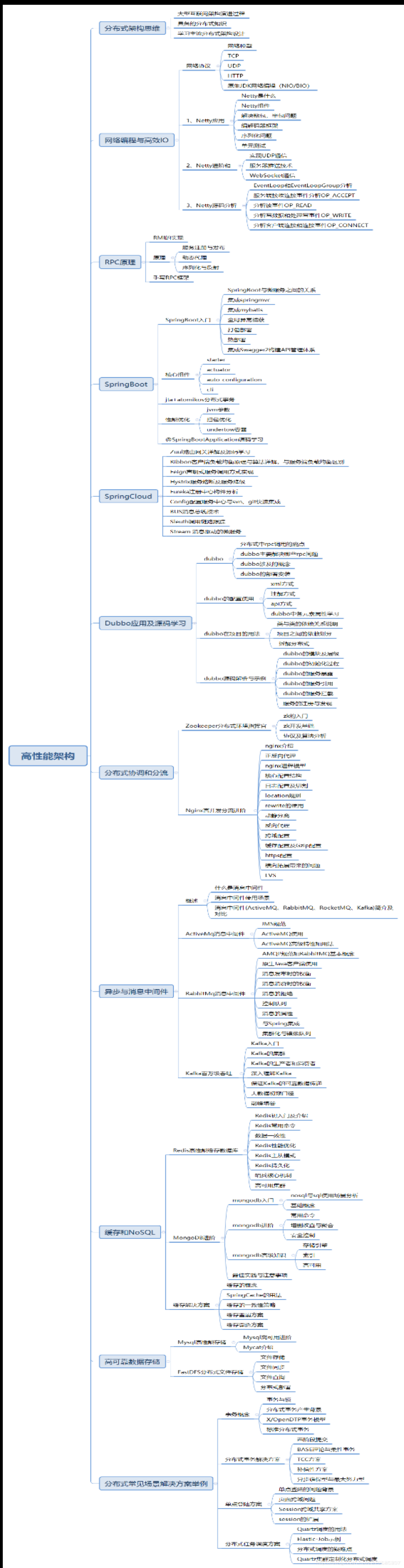
Kafka 学习笔记

分布式架构学习路线：

Edwin Xu


高性能架构
分布式架构思维
大型互联网架构演进过程
具备的分布式知识
学习丰富分布式架构设计
网络编程与高效IO
网络协议
网络模型
TCP
UDP
HTTP
原生JDK网络编程（NIO/BIO）
1、Netty应用
Netty是什么
Netty组件
解决粘包、半包问题
编解码器框架
序列化问题
单元测试
2、Netty进阶和
实现UDP通信
服务器推送技术
WebSocket通信
3、Netty源码分析
EventLoop和EventLoopGroup分析
服务端接收连接事件分析OP_ACCEPT
分析读事件OP_READ
分析写数据和处理写事件OP_WRITE
分析客户端连接和连接事件OP_CONNECT
RPC原理
RMI的实现
原理
服务注册与发布
动态代理
序列化与反射
手写RPC框架
SpringBoot
SpringBoot入门
SpringBoot与微服务之间的关系
集成springmvc
集成mybatis
全局异常捕获
打包部署
热部署
集成Swagger2构建API管理体系
核心组件
starter
actuator
auto configuration
cli
jta+atomikos分布式事务
性能优化
jvm参数
扫包优化
undertow容器
@SpringBootApplication源码学习
SpringCloud
Zuul路由网关详解及源码学习
Ribbon客户端负载均衡原理与算法详解，与服务端负载均衡区别
Feign声明式服务调用方式实现
Hystrix服务熔断及服务降级
Eureka注册中心构件分析
Config配置服务中心与svn、git快速集成
BUS消息总线技术
Sleuth调用链路跟踪
Stream 消息驱动的微服务
Dubbo应用及源码学习
dubbo
分布式中rpc调用的痛点
dubbo主要解决哪些rpc问题
dubbo涉及的概念
dubbo的部署安装
dubbo的配置使用
xml方式
注解方式
api方式
dubbo中各元素属性学习
dubbo在项目的用法
类与类的依赖关系说明
项目之间的依赖划分
拆解分布式
dubbo源码解析与示例
dubbo的模块及层级
dubbo的初始化过程
dubbo的服务暴露
dubbo的服务引用
dubbo的服务拦截
服务的注册与发现
分布式协调和分流
Zookeeper分布式环境指挥官
zk的入门
zk开发基础
协议及算法分析
Nginx高并发分流进阶
nginx介绍
正反向代理
nginx进程模型
核心配置结构
日志配置及切割
location规则
rewrite的使用
动静分离
反向代理
跨域配置
缓存配置及Gzip配置
https配置
横向拓展带来的问题
LVS
异步与消息中间件
概述
什么是消息中间件
消息中间件使用场景
消息中间件(ActiveMQ、RabbitMQ、RocketMQ、Kafka)简介及对比
ActiveMq消息中间件
JMS规范
ActiveMQ使用
ActiveMQ高级特性和用法
RabbitMq消息中间件
AMQP规范和RabbitMQ基本概念
原生Java客户端使用
消息发布时的权衡
消息消费时的权衡
消息的拒绝
控制队列
消息的属性
与Spring集成
集群化与镜像队列
Kafka百万级吞吐
Kafka入门
Kafka的集群
Kafka的生产者和消费者
深入理解Kafka
保证Kafka的可靠数据传递
大数据的入门
调优指南
缓存和NoSQL
Redis高性能缓存数据库
Redis的入门及介绍
Redis常用命令
数据一致性
Redis性能优化
Redis主从模式
Redis持久化
哨兵核心机制
高可用集群
MongoDB进阶
mongodb入门
nosql与sql使用场景分析
基础概念
mongodb进阶
常用命令
增删改查与聚合
安全控制
mongodb高级知识
存储引擎
索引
高可用
最佳实践与注意事项
缓存解决方案
缓存的概念
SpringCache的用法
缓存的一致性策略
缓存雪崩方案
缓存穿透方案
高可靠数据存储
Mysql高性能存储
Mysql高可用进阶
Mycat介绍
FastDFS分布式文件存储
文件存储
文件同步
文件查询
分布式部署
分布式常见场景解决方案举例
事务概念
事务与锁
分布式事务产生背景
X/OpenDTP事务模型
标准分布式事务
分布式事务解决方案
两阶段提交
BASE理论与柔性事务
TCC方案
补偿性方案
异步确保型与最大努力型
单点登陆方案
单点登陆的问题背景
页面跨域问题
Session跨域共享方案
session的扩展
分布式任务调度方案
Quartz调度的用法
Elastic-Job示例
分布式调度的疑难点
Quartz集群定制化分布式调度

Edwin Xu

消息中间件 MQ

消息队列 MQ(Message Queue)已经逐渐成为企业 IT 系统内部通信的核心手段。它具有低耦合、可靠投递、广播、流量控制、最终一致性等一系列功能，成为**异步 RPC** 的主要手段之一。当今市面上有很多主流的消息中间件，如老牌的 ActiveMQ、RabbitMQ，炙手可热的 Kafka，阿里巴巴自主开发 RocketMQ 等。

MQ 的组成

- Broker：消息服务器，作为 Server，提供消息核心服务。
- Producer：消息生产者，业务的发起方，负责生产消息传输给 broker
- Consumer：消息消费者，业务的处理方，负责从 broker 获取消息并进行业务逻辑处理。
- Topic：主题。发布-订阅模式下的消息统一汇集地，不同的 producer 向 topic 发送消息，由 MQ 服务器分发到不同的订阅者，实现消息广播。
- Queue：队列，producer 向队列发送消息，consumer 从队列中接受消息。
- Message：消息体。根据不同通信协议定义的固定格式进行编码的数据包，来封装业务数据，实现消息的传输

消息中间件模式分类

- 点对点

  PTP：Point-To-Point

  使用 queue 作为通信载体

  消息生产者生产消息发送到 queue 中，然后消息消费者从 queue 中取出并且消费消息。

  消息被消费以后，queue 中不再存储，所以消息消费者不可能消费到已经被消费的消息。 Queue 支持存在多个消费者，但是对一个消息而言，只会有一个消费者可以消费。
- 发布/订阅

  Pub/Sub 发布订阅（广播）：使用 topic 作为通信载体

  消息生产者（发布）将消息发布到 topic 中，同时有多个消息消费者（订阅）消费该消息。和点对点方式不同，发布到 topic 的消息会被所有订阅者消费。

  queue 实现了负载均衡，将 producer 生产的消息发送到消息队列中，由多个消费者消费。但一个消息只能被一个消费者接受，当没有消费者可用时，这个消息会被保存直到有一个可用的消费者。

  topic 实现了发布和订阅，当你发布一个消息，所有订阅这个 topic 的服务都能得到这个消息，所以从 1 到 N 个订阅者都能得到一个消息的拷贝。

消息中间件的优势

1. 系统解耦：交互系统之间没有直接的调用关系，只是通过消息传输，故系统侵入性不强，耦合度

低。
2. 提高系统响应时间
3. 为大数据处理架构提供服务：大数据的背景下，消息队列还与实时处理架构整合，为数据处理提供性能支持。
4. Java 消息服务——JMS
   Java 消息服务（Java Message Service，JMS）应用程序接口是一个 Java 平台中关于面向消息中间件（MOM）的 API，用于在两个应用程序之间，或分布式系统中发送消息，进行异步通信。JMS 中的 P2P 和 Pub/Sub 消息模式：点对点（point to point， queue）与发布订阅（publish/subscribe，topic）最初是由 JMS 定义的。这两种模式主要区别或解决的问题就是发送到队列的消息**能否重复消费(多订阅)**。

消息中间件应用场景

1. 异步通信：有些业务不想也不需要立即处理消息。消息队列提供了异步处理机制，允许用户把一个消息放入队列，但并不立即处理它。想向队列中放入多少消息就放多少，然后在需要的时候再去处理它们。
2. 解耦
3. 冗余
4. 扩展性
5. 过载保护
6. 可恢复性
7. 顺序保证
8. 缓冲
9. 数据流处理

消息中间件常用协议

- AMQP 协议
  AMQP 即 Advanced Message Queuing Protocol,一个提供统一消息服务的应用层标准高级消息队列协议,是应用层协议的一个开放标准,为面向消息的中间件设计。基于此协议的客户端与消息中间件可传递消息，并不受客户端/中间件不同产品，不同开发语言等条件的限制。
  优点：可靠、通用
- MQTT
  MQTT（Message Queuing Telemetry Transport，消息队列遥测传输）是 IBM 开发的一个即时通讯协议，有可能成为物联网的重要组成部分。该协议支持所有平台，几乎可以把所有联网物品和外部连接起来，被用来当做传感器和致动器（比如通过 Twitter 让房屋联网）的通信协议。
  优点：格式简洁、占用带宽小、移动端通信、PUSH、嵌入式系统
- STOMP
  STOMP（Streaming Text Orientated Message Protocol）是流文本定向消息协议，是一种为 MOM(Message Oriented Middleware，面向消息的中间件)设计的简单文本协议。STOMP 提供一个可互操作的连接格式，允许客户端与任意 STOMP 消息代理（Broker）进行交互。
  优点：命令模式（非 topic\queue 模式）

- XMPP

  XMPP（可扩展消息处理现场协议，Extensible Messaging and Presence Protocol）是基于可扩展标记语言（XML）的协议，多用于即时消息（IM）以及在线现场探测。适用于服务器之间的准即时操作。核心是基于 XML 流传输，这个协议可能最终允许因特网用户向因特网上的其他任何人发送即时消息，即使其操作系统和浏览器不同。

  优点：通用公开、兼容性强、可扩展、安全性高，但 XML 编码格式占用带宽大

- 其他基于 TCP/IP 自定义的协议

  有些特殊框架（如 **redis、kafka、zeroMq** 等）根据自身需要未严格遵循 MQ 规范，而是基于 TCP\IP 自行封装了一套协议，通过网络 socket 接口进行传输，实现了 MQ 的功能。

发布-订阅 设计模式

常见消息中间件 MQ 介绍

- RocketMQ

  阿里系下开源的一款分布式、队列模型的消息中间件，原名 Metaq，3.0 版本名称改为 RocketMQ，是阿里参照 kafka 设计思想使用 java 实现的一套 mq。同时将阿里系内部多款 mq 产品（Notify、metaq）进行整合，只维护核心功能，去除了所有其他运行时依赖，保证核心功能最简化，在此基础上配合阿里上述其他开源产品实现不同场景下 mq 的架构，目前主要多用于订单交易系统。

  具有以下特点：

  - 能够保证严格的消息顺序
  - 提供针对消息的过滤功能
  - 提供丰富的消息拉取模式
  - 高效的订阅者水平扩展能力
  - 实时的消息订阅机制
  - 亿级消息堆积能力

  官方提供了一些不同于 kafka 的对比差异：

  https://rocketmq.apache.org/docs/motivation/

- RabbitMQ

  使用 Erlang 编写的一个开源的消息队列，本身支持很多的协议：AMQP，XMPP，SMTP,STOMP，也正是如此，使的它变的非常**重量级**，更适合于企业级的开发。同时实现了 Broker 架构，核心思想是生产者不会将消息直接发送给队列，消息在发送给客户端时先在中心队列排队。对路由(Routing)，负载均衡(Load balance)、数据持久化都有很好的支持。多用于进行企业级的 ESB 整合。

- ActiveMQ

  Apache 下的一个子项目。使用 Java 完全支持 JMS1.1 和 J2EE 1.4 规范的 JMS Provider 实现，少量代码就可以高效地实现高级应用场景。可插拔的传输协议支持，比如：in-VM，TCP，SSL，NIO，UDP，multicast，JGroups and JXTA transports。RabbitMQ、ZeroMQ、ActiveMQ 均支持常用的多种语言客户端 C++、Java、.Net,、Python、 Php、 Ruby 等。

- Redis

  使用**C语言**开发的一个Key-Value的NoSQL数据库,开发维护很活跃,虽然它是一个Key-Value数据库存储系统，但它本身支持MQ功能，所以完全**可以当做一个轻量级的队列服务来使用**。对于RabbitMQ和Redis的入队和出队操作，各执行100万次，每10万次记录一次执行时间。测试数据分为128Bytes、512Bytes、1K和10K四个不同大小的数据。实验表明：入队时，当数据比较小时Redis的性能要高于RabbitMQ，而如果数据大小超过了10K，Redis则慢的无法忍受；出队时，无论数据大小，Redis都表现出非常好的性能，而RabbitMQ的出队性能则远低于Redis。

- Kafka

  Apache下的一个子项目，使用scala实现的一个高性能分布式Publish/Subscribe消息队列系统，具有以下特性：

  1. 快速持久化：通过磁盘顺序读写与零拷贝机制，可以在O(1)的系统开销下进行消息持久化；
  2. 高吞吐：在一台普通的服务器上既可以达到10W/s的吞吐速率；
  3. 高堆积：支持topic下消费者较长时间离线，消息堆积量大；
  4. 完全的分布式系统：Broker、Producer、Consumer都原生自动支持分布式，依赖zookeeper自动实现复杂均衡；
  5. 支持Hadoop数据并行加载：对于像Hadoop的一样的日志数据和离线分析系统，但又要求实时处理的限制，这是一个可行的解决方案。

- ZeroMQ

  号称**最快的消息队列**系统，专门为**高吞吐量/低延迟**的场景开发，在**金融界**的应用中经常使用，偏重于实时数据通信场景。ZMQ能够实现RabbitMQ不擅长的高级/复杂的队列，但是开发人员需要自己组合多种技术框架，开发成本高。因此ZeroMQ具有一个独特的非中间件的模式，更像一个socket library，你不需要安装和运行一个消息服务器或中间件，因为你的应用程序本身就是使用ZeroMQ API完成逻辑服务的角色。但是ZeroMQ仅提供非持久性的队列，如果down机，数据将会丢失。如：Twitter的Storm中使用ZeroMQ作为数据流的传输。

  ZeroMQ套接字是与传输层无关的：ZeroMQ套接字对所有传输层协议定义了统一的API接口。默认支持 进程内(inproc) ，进程间(IPC) ，多播，TCP协议，在不同的协议之间切换只要简单的改变连接字符串的前缀。可以在任何时候以最小的代价从进程间的本地通信切换到分布式下的TCP通信。ZeroMQ在背后处理连接建立，断开和重连逻辑。

  特性：

  1. 无锁的队列模型：对于跨线程间的交互（用户端和session）之间的数据交换通道pipe，采用无锁的队列算法CAS；在pipe的两端注册有异步事件，在读或者写消息到pipe的时，会自动触发读写事件。
  2. 批量处理的算法：对于批量的消息，进行了适应性的优化，可以批量的接收和发送消息。
  3. 多核下的线程绑定，无须CPU切换：区别于传统的多线程并发模式，信号量或者临界区，zeroMQ充分利用多核的优势，每个核绑定运行一个工作者线程，避免多线程之间的CPU切换开销。

主要消息中间件的比较：

A	B	C	D	E
特性	ActiveMQ	RabbitMQ	RocketMQ	Kafka
PRODUCER-COMSUMER	支持	支持	支持	支持
PUBLISH-SUBSCRIBE	支持	支持	支持	支持
REQUEST-REPLY	支持	支持		
API完备性	高	高	高	高
多语言支持	支持，JAVA优先	语言无关	只支持JAVA	支持，java优先
单机吞吐量	万级	万级	万级	十万级
消息延迟		微秒级	毫秒级	毫秒级
可用性	高（主从）	高（主从）	非常高（分布式）	非常高（分布式）
消息丢失	低	低	理论上不会丢失	理论上不会丢失
消息重复		可控制		理论上会有重复
文档的完备性	高	高	高	高
提供快速入门	有	有	有	有
首次部署难度		低		中
社区活跃度	高	高	中	高
商业支持	无	无	阿里云	无
成熟度	成熟	成熟	比较成熟	成熟日志领域
特点	功能齐全，被大 量开源项目使用	由于Erlang语言的并发能 力，性能很好	各个环节分 布式扩展设计，主从 HA；支持上万个队列；多 种消费模式；性能很好	
支持协议	OpenWire、 STOMP、 REST、XMPP、AMQP	AMQP	自己定义的一 套(社区提供 JMS--不成熟)	
持久化	内存、文件、数据库	内存、文件	磁盘文件	
事务	支持	支持	支持	
负载均衡	支持	支持	支持	
管理界面	一般	好	有web console实现	
部署方式	独立、嵌入	独立	独立	
评价	优点：成熟的产品，已经在很多公 司得到应用（非大规模场景）。有 较多的文档。各种协议支持较好， 有多重语言的成熟的客户端； 缺点：根据其他 用户反馈，会出莫 名其妙的问题，切会丢 失消息。其 重心放到 activemq6.0 产品—apollo 上去了，目前社区不活跃，且对5.x 维护较少； Activemq 不适合用于上 千个队列的应用场景。	优点：由于erlang语言的特 性，mq性能较好；管理界面 较丰富，在互联网公司也有 较大规模的应用；支持amqp 系诶，有多中语言且支持 amqp的客户端可用； 缺点：erlang语言难度较 大。集群不支持动态扩展。	优点：模型简单，接口易用（JMS的接口很多 场合并不太实用）。在阿里大规模应用。目前 支付宝中的余额宝等新兴产品均使用 rocketmq 。集群规模大概在50台左右，单日处理消息上 百亿；性能非常好，可以大量堆积消息在 broker中；支持多种消费，包括集群消费、广 播消费等。开发度较活跃，版本更新很快。 缺点：产品较新文档比较缺乏。没有在mq核心 中去实现JMS等接口，对已有系统而言不能兼 容。阿里内部还有一套未开源的MQAPI，这一 层API可以将上层应用和下层MQ的实现解耦 （阿里内部有多个mq的实现，如notify、 metaq1.x，metaq2.x，rocketmq等），使得下 面mq可以很方便的进行切换和升级而对应用无 任何影响，目前这一套东西未开源。 http://blog.csdn.net/oMaverick1	

官方教程学习

Site: https://kafka.apachecn.org/

卡夫卡
官方文档是最好的教程

介绍

介绍

Apache Kafka 是一个**分布式流处理平台(Distributed Streaming Platform).**

流处理平台有以下三种特性:
1. 可以发布-订阅流式的记录: 类似消息队列。
2. **存储**流式的记录, 有较好容错性
3. 能在流式记录产生时就处理。

可以应用于两大类别的应用:
1. 构造实时流数据管道: 可以在系统或应用之间可靠地获取数据(相当于 Message Queue)
2. 构建实时流式应用程序: 对这些流数据进行转换或者影响(即流处理, 通过 Stream topic 和 topic 之间内部进行变化)

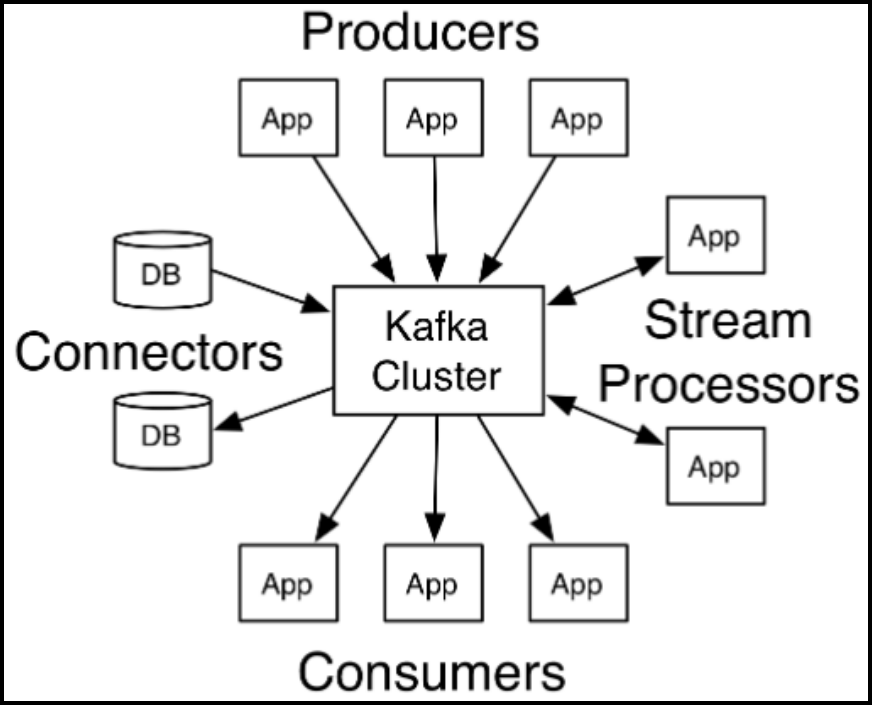
基本的认识:
1. Kafka 作为一个集群, 运行在一台或者多台服务器上。
2. Kafka 通过 topic 对存储的流数据进行分类。
3. 每条记录中包含一个 key、一个 value 和一个 timestamp

四个核心 API

1. The Producer API : 允许一个应用程序发布一串流式的数据到一个或者多个 Kafka topic。
2. The Consumer API : 允许一个应用程序订阅一个或多个 topic , 并且对发布给他们的流式数据进行处理。
3. The Streams API: 允许一个应用程序作为一个流处理器, 消费一个或者多个 topic 产生的输

入流，然后生产一个输出流到一个或多个 topic 中去，在输入输出流中进行有效的转换。

4. The Connector API ： 允许构建并运行可重用的生产者或者消费者，将 Kafka topics 连接到已存在的应用程序或者数据系统。比如，连接到一个关系型数据库，捕捉表（table）的所有变更内容。

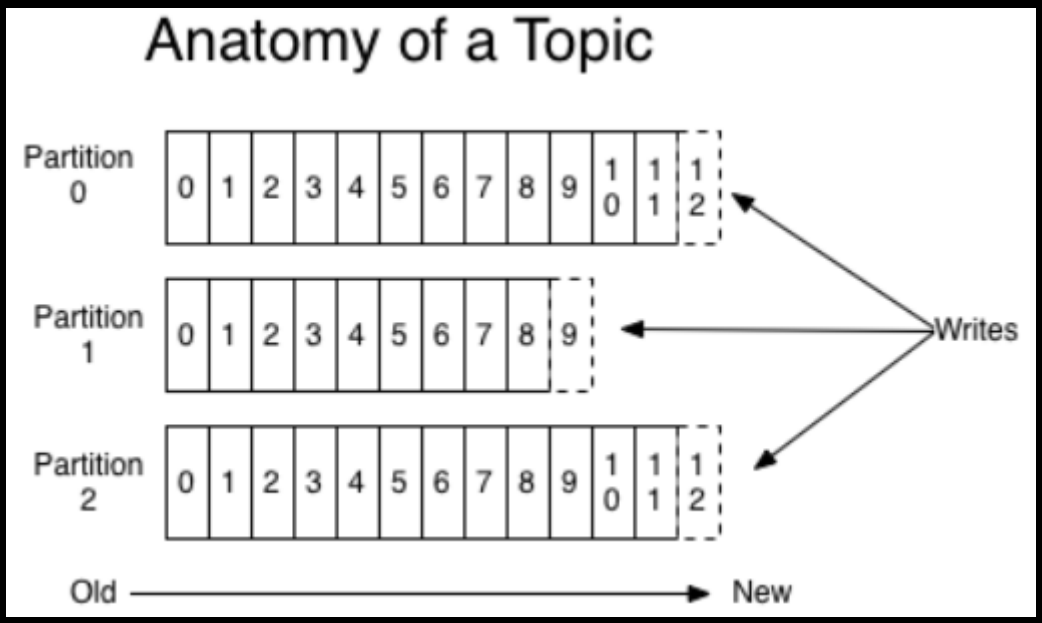


在 Kafka 中，客户端和服务器使用一个简单、高性能、支持多语言的 TCP 协议.此协议版本化并且向下兼容老版本， 我们为 Kafka 提供了 Java 客户端，也支持许多其他语言的客户端。

Topics 和日志

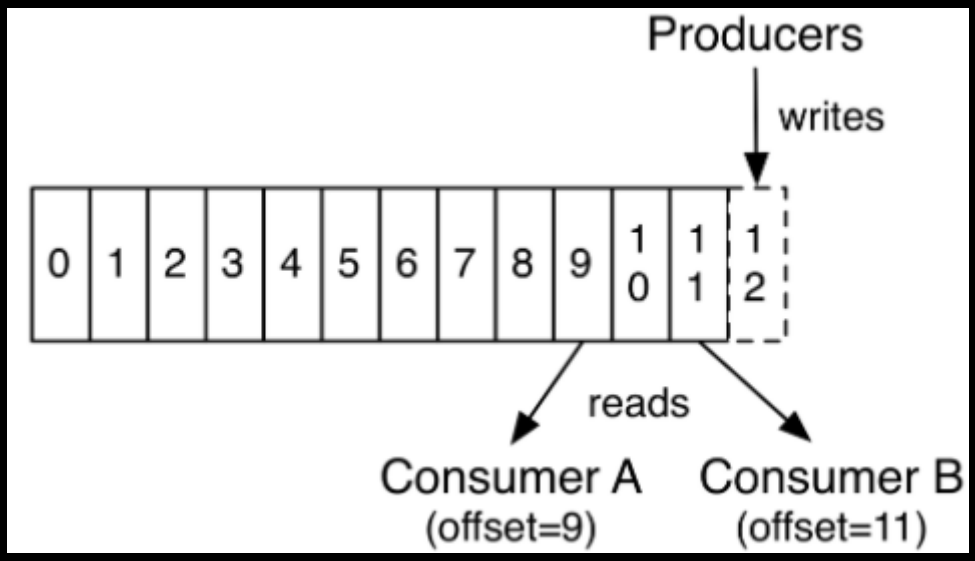
Kafka 的核心概念： **提供一串流式的记录— topic**
Topic 就是数据主题，是数据记录发布的地方,可以用来区分业务系统。Kafka 中的 **Topics 总是多订阅者模式**，一个 topic 可以拥有一个或者多个消费者来订阅它的数据。

对于每一个 topic， Kafka 集群都会维持一个分区日志，如下所示：



每个分区都是有序且顺序不可变的记录集，并且不断地追加到结构化的 commit log 文件。分区中的每一个记录都会分配一个 id 号来表示顺序，我们称之为 offset，offset 用来唯一的标识分区中每一条记录。

Kafka 集群保留所有发布的记录—无论他们是否已被消费—并通过一个可配置的参数——保留期限来控制. 举个例子， 如果保留策略设置为 2 天，一条记录发布后两天内，可以随时被消费，两天过后这条记录会被抛弃并释放磁盘空间。Kafka 的性能和数据大小无关，所以长时间存储数据没有什么问题.



事实上，在每一个消费者中唯一保存的元数据是 offset（偏移量）即消费在 log 中的位置.偏移量由消费者所控制:通常在读取记录后，消费者会以线性的方式增加偏移量，但是实际上，由于这个位置由消费者控制，所以消费者可以采用任何顺序来消费记录。例如，一个消费者可以重置到一个旧的偏移量，从而重新处理过去的数据；也可以跳过最近的记录，从"现在"开始消费。

这些细节说明 Kafka 消费者是非常廉价的—消费者的增加和减少，对集群或者其他消费者没有多大的影响。比如，你可以使用命令行工具，对一些 topic 内容执行 tail 操作，并不会影响已存在的消费者消费数据。

日志中的 partition（分区）有以下几个用途。第一，当日志大小超过了单台服务器的限制，允许日志进行扩展。每个单独的分区都必须受限于主机的文件限制，不过一个主题可能有多个分区，因此可以处理无限量的数据。第二，可以作为并行的单元集—关于这一点，更多细节如下

分布式

日志的分区 partition （分布）在 Kafka 集群的服务器上。每个服务器在处理数据和请求时，共享这些分区。每一个分区都会在已配置的服务器上进行备份，确保容错性.

每个分区都有**一台 server 作为 “leader”**，零台或者多台 server 作为 follwers 。leader server 处理一切对 partition （分区）的读写请求，而 follwers 只需被动的同步 leader 上的数据。当 leader 宕机了，followers 中的一台服务器会自动成为新的 leader。每台 server 都会成为某些分区的 leader 和某些分区的 follower，因此集群的负载是平衡的。

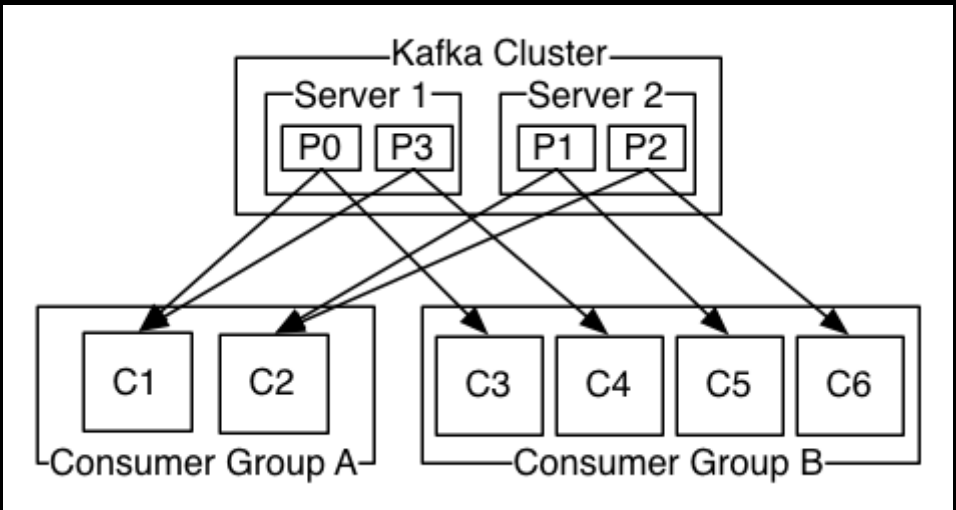
生产者

生产者可以将数据发布到所选择的 topic（主题）中。生产者负责**将记录分配到 topic 的哪一个 partition（分区）**中。可以使用循环的方式来简单地实现负载均衡，也可以根据某些语义分区函数(例如：记录中的 key)来完成。

消费者

消费者使用一个 **消费组** 名称来进行标识，发布到 topic 中的每条记录被分配给订阅消费组中的一个消费者实例.消费者实例可以分布在多个进程中或者多个机器上。

如果所有的消费者实例在同一消费组中，消息记录会负载平衡到每一个消费者实例.
如果所有的消费者实例在不同的消费组中，每条消息记录会广播到所有的消费者进程.



如图，这个 Kafka 集群有两台 server 的，四个分区(p0-p3)和两个消费者组。消费组 A 有两个消费者，消费组 B 有四个消费者。
通常情况下，每个 topic 都会有一些消费组，一个消费组对应一个"逻辑订阅者"。一个消费组由许多消费者实例组成，便于扩展和容错。这就是发布和订阅的概念，只不过订阅者是一组消费者而不是单个的进程。
在 Kafka 中实现消费的方式是将日志中的分区划分到每一个消费者实例上，以便在任何时间，每个实例都是分区唯一的消费者。维护消费组中的消费关系由 Kafka 协议动态处理。如果新的实例加入组，他们将从组中其他成员处接管一些 partition 分区;如果一个实例消失，拥有的分区将被分发到剩余的实例。
Kafka 只保证分区内的记录是有序的，而不保证主题中不同分区的顺序。每个 partition 分区按照 key 值排序足以满足大多数应用程序的需求。但如果你需要总记录在所有记录的上面，可使用仅有一个分区的主题来实现，这意味着每个消费者组只有一个消费者进程。

保证

1. 生产者发送到特定 topic partition 的消息将**按照发送的顺序处理**。 也就是说，如果记录 M1 和记录 M2 由相同的生产者发送，并先发送 M1 记录，那么 M1 的偏移比 M2 小，并在日志中较早出现
2. 一个消费者实例按照日志中的顺序查看记录.
3. 对于具有 N 个副本的主题，我们最多容忍 N-1 个服务器故障，从而保证不会丢失任何提交到日志中的记录.

Kafka 作为消息系统

Kafka streams 的概念与传统的企业消息系统相比如何?

传统的消息系统有两个模块: 队列 和 发布-订阅。 在队列中，消费者池从 server 读取数据，每条记录被池子中的一个消费者消费； 在发布订阅中，记录被广播到所有的消费者。两者均有优缺点。队列的优点在于它允许你将处理数据的过程分给多个消费者实例，使你可以扩展处理过程。 不好的

是，队列不是多订阅者模式的——一旦一个进程读取了数据，数据就会被丢弃。 而发布-订阅系统允许你广播数据到多个进程，但是无法进行扩展处理，因为每条消息都会发送给所有的订阅者。

消费组在 Kafka 有两层概念。在队列中，消费组允许你将处理过程分发给一系列进程(消费组中的成员)。 在发布订阅中，Kafka 允许你将消息广播给多个消费组。

Kafka 的优势在于每个 topic 都有以下特性—**可以扩展处理并且允许多订阅者模式**(队列+发布订阅)—不需要只选择其中一个.

Kafka 相比于传统消息队列还具有更严格的顺序保证
传统队列在服务器上保存有序的记录，如果多个消费者消费队列中的数据， 服务器将按照存储顺序输出记录。 虽然服务器按顺序输出记录，但是记录被异步传递给消费者， 因此记录可能会无序的到达不同的消费者。这意味着在并行消耗的情况下， 记录的顺序是丢失的。因此消息系统通常使用“**唯一消费者**”的概念，即只让一个进程从队列中消费， 但这就意味着不能够并行地处理数据。

Kafka 设计的更好。**topic 中的 partition 是一个并行的概念**。 Kafka 能够为一个消费者池提供顺序保证和负载平衡，是通过将 topic 中的 partition 分配给消费者组中的消费者来实现的， 以便**每个分区由消费组中的一个消费者消耗**。通过这样，我们能够确保**消费者是该分区的唯一读者**，并按顺序消费数据。 众多分区保证了多个消费者实例间的负载均衡。但请注意，消费者组中的消费者实例个数不能超过分区的数量。

Kafka 作为存储系统

数据写入 Kafka 后被写到磁盘，并且进行备份以便容错。直到完全备份，Kafka 才让生产者认为完成写入，即使写入失败 Kafka 也会确保继续写入
Kafka 使用磁盘结构，具有很好的扩展性—50kb 和 50TB 的数据在 server 上表现一致。

可以存储大量数据，并且可通过客户端控制它读取数据的位置，您可认为 Kafka 是一种高性能、低延迟、具备日志存储、备份和传播功能的分布式文件系统。

Kafka 用做流处理

Kafka 流处理不仅仅用来读写和存储流式数据，它最终的目的是为了能够进行**实时的流处理**。
在 Kafka 中，流处理器不断地从输入的 topic 获取流数据，处理数据后，再不断生产流数据到输出的 topic 中去。

批处理

像 HDFS 这样的分布式文件系统可以存储用于批处理的静态文件。 一个系统如果可以存储和处理历史数据是非常不错的
传统的企业消息系统允许处理订阅后到达的数据。以这种方式来构建应用程序，并用它来处理即将到达的数据。
Kafka 结合了上面所说的两种特性。作为一个流应用程序平台或者流数据管道，这两个特性，对于 Kafka 来说是至关重要的。

Quickstart

1. 下载：官网 or 清华镜像站
2. 启动服务器
   Kafak 使用 zookeeper，所以先启动一个 zookeeper 服务器。
   不过下载的包里面也包含 zk，可以直接启动：
   bin/zookeeper-server-start.sh config/zookeeper.properties
   然后启动 kafka：
   bin/kafka-server-start.sh config/server.properties

3. 创建一个 topic
   创建一个名为“test”的 topic，它有一个分区和一个副本：

```
bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic test
```

   运行 list 命令查看这个 topic
   bin/kafka-topics.sh --list --zookeeper localhost:2181

4. 发送消息
   Kafka 自带一个命令行客户端，它从文件或标准输入中获取输入，并将其作为 message（消息）发送到 Kafka 集群。默认情况下，每行将作为单独的 message 发送。
   运行 producer，然后在控制台输入一些消息以发送到服务器。

```
> bin/kafka-console-producer.sh --broker-list localhost:9092 --topic test
This is a message
This is another message
```

5. 启动一个 consumer
   Kafka 还有一个命令行 consumer（消费者），将消息转储到标准输出。

```
> bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic test --from-beginning
This is a message
This is another message
```

6. 设置多代理集群
   到目前为止，我们一直在使用单个代理，这并不好玩。对 Kafka 来说，单个代理只是一个大小为一的集群，除了启动更多的代理实例外，没有什么变化。 为了深入了解它，让我们把集群扩展到三个节点
   首先，为每个代理创建一个配置文件：

```
> cp config/server.properties config/server-1.properties
> cp config/server.properties config/server-2.properties
现在编辑这些新文件并设置如下属性:
```

编辑这些新文件并设置如下属性:

```
config/server-1.properties:
    broker.id=1
    listeners=PLAINTEXT://:9093
    log.dir=/tmp/kafka-logs-1

config/server-2.properties:
    broker.id=2
    listeners=PLAINTEXT://:9094
    log.dir=/tmp/kafka-logs-2
```

broker.id 属性是集群中每个节点的名称，这一名称是唯一且永久的。我们必须重写端口和日志目录，因为我们在同一台机器上运行这些，我们不希望所有的代理尝试在同一个端口注册，或者覆盖彼此的数据。

我们已经建立 Zookeeper 和一个单节点了，现在我们只需要启动两个新的节点:

```
> bin/kafka-server-start.sh config/server-1.properties &
...
> bin/kafka-server-start.sh config/server-2.properties &
...
```

现在创建一个副本为 3 的新 topic:

```
bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 3 --partitions 1 --topic my-replicated-topic
```

不写了，自己操作

7. 使用 Kafka Connect 来导入/导出数据

可以使用 Kafka Connect 来导入或导出数据，而不是写自定义的集成代码。

8. 使用 Kafka Streams 来处理数据

应用

消息:

Kafka 很好地替代了传统的 message broker（消息代理）。 Message brokers 可用于各种场合（如将数据生成器与数据处理解耦，缓冲未处理的消息等）。 与大多数消息系统相比，Kafka 拥有更好的吞吐量、内置分区、具有复制和容错的功能，这使它成为一个非常理想的大型消息处理应用。在这方面，Kafka 可以与传统的消息传递系统（ActiveMQ 和 RabbitMQ）相媲美。

跟踪网站活动:

网站活动（浏览网页、搜索或其他的用户操作）将被发布到中心 topic，其中每个活动类型有一个 topic。 这些订阅源提供一系列用例，包括实时处理、实时监视、对加载到 Hadoop 或离线数据仓库系统的数据进行离线处理和报告等。

(热搜这种不就可以使用 kafka 流处理吗？)

度量:

Kafka 通常用于监控数据。这涉及到从分布式应用程序中汇总数据，然后生成可操作的集中数据源。

日志聚合：
许多人使用 Kafka 来替代日志聚合解决方案。 日志聚合系统通常从服务器收集物理日志文件，并将其置于一个中心系统（可能是文件服务器或 HDFS）进行处理。 Kafka 从这些日志文件中提取信息，并将其抽象为一个更加清晰的消息流。 这样可以实现更低的延迟处理且易于支持多个数据源及分布式数据的消耗。 与 Scribe 或 Flume 等以日志为中心的系统相比，Kafka 具备同样出色的性能、更强的耐用性（因为复制功能）和更低的端到端延迟。

流处理：
许多 Kafka 用户通过管道来处理数据，有多个阶段： 从 Kafka topic 中消费原始输入数据，然后聚合，修饰或通过其他方式转化为新的 topic， 以供进一步消费或处理。 例如，一个推荐新闻文章的处理管道可以从 RSS 订阅源抓取文章内容并将其发布到“文章”topic；然后对这个内容进行标准化或者重复的内容， 并将处理完的文章内容发布到新的 topic；最终它会尝试将这些内容推荐给用户。 这种处理管道基于各个 topic 创建实时数据流图。从 0.10.0.0 开始，在 Apache Kafka 中，Kafka Streams 可以用来执行上述的数据处理，它是一个轻量但功能强大的流处理库。除 Kafka Streams 外，可供替代的开源流处理工具还包括 Apache Storm 和 Apache Samza.

采集日志：

提交日志：

Spring-kafka

https://docs.spring.io/spring-kafka/docs/2.2.6.RELEASE/reference/html/#preface

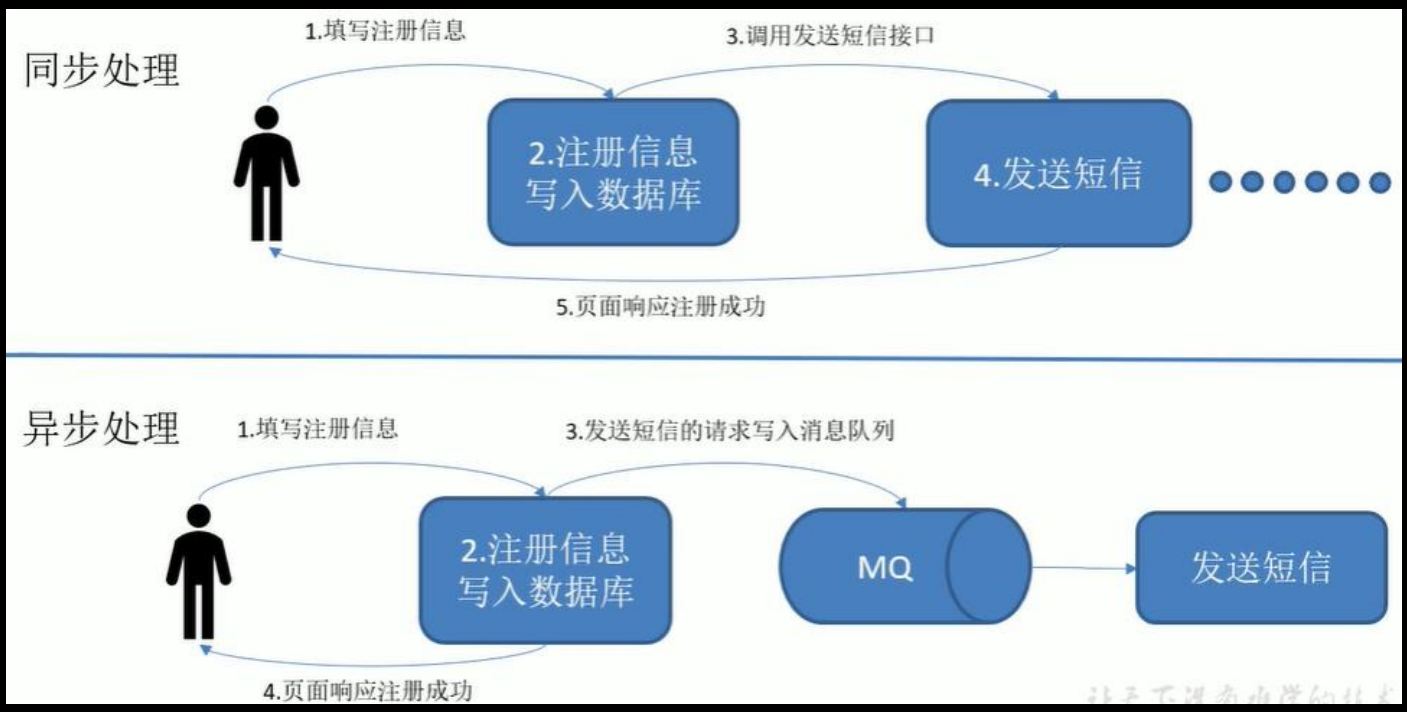
视频学习

https://www.bilibili.com/video/BV1a4411B7V9?from=search&seid=14135120435675372102

kafka 概述

是一个**分布式**的基于发布/订阅模式的消息队列，主要应用于大数据实时处理领域

应用场景

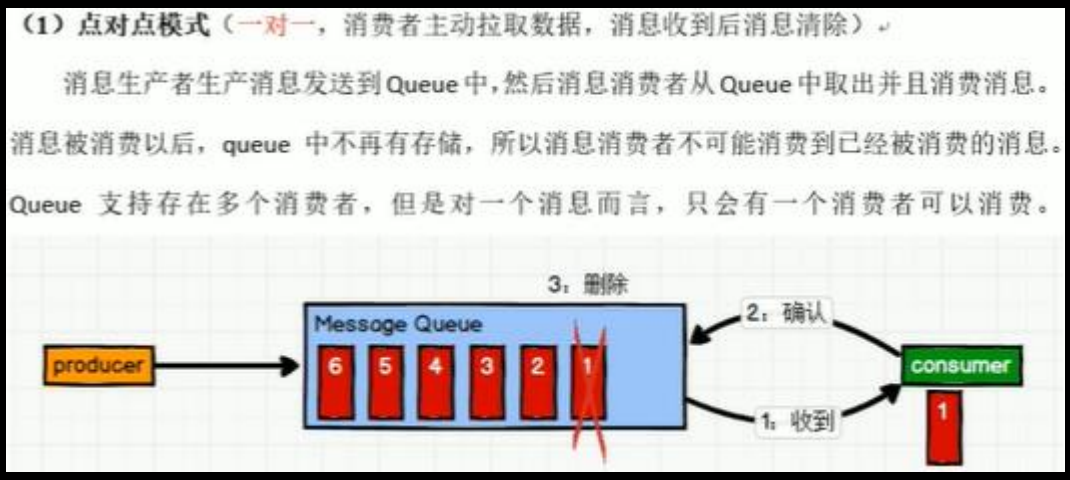


好处：

1. 解耦：分离了两个过程
2. 可恢复性：系统的一部分崩溃时不会影响整个系统
3. 缓冲：生产大于消费时，可以缓冲，
4. 灵活性、峰值处理
5. 异步通信

两种模式

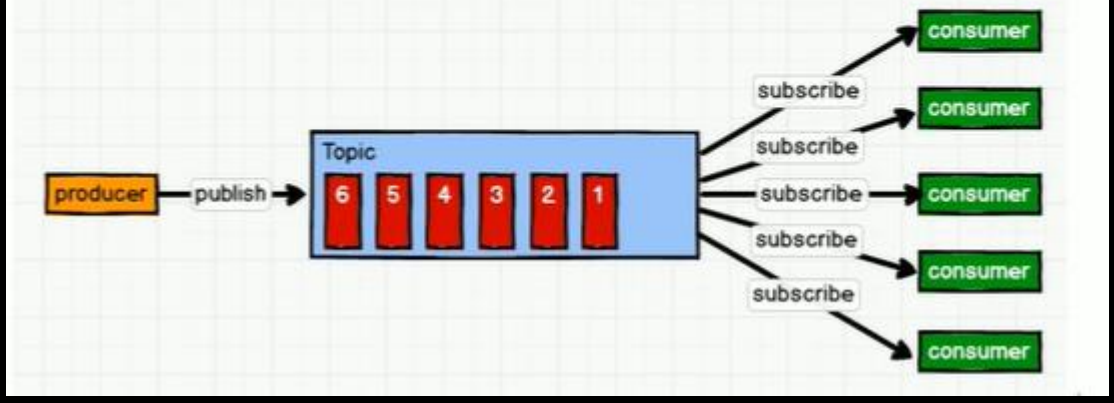
1. 点对点模式



2. 发布/订阅模式

（2）发布/订阅模式（一对多，消费者消费数据之后不会清除消息）

消息生产者（发布）将消息发布到 topic 中，同时有多个消息消费者（订阅）消费该消息。和点对点方式不同，发布到 topic 的消息会被所有订阅者消费。

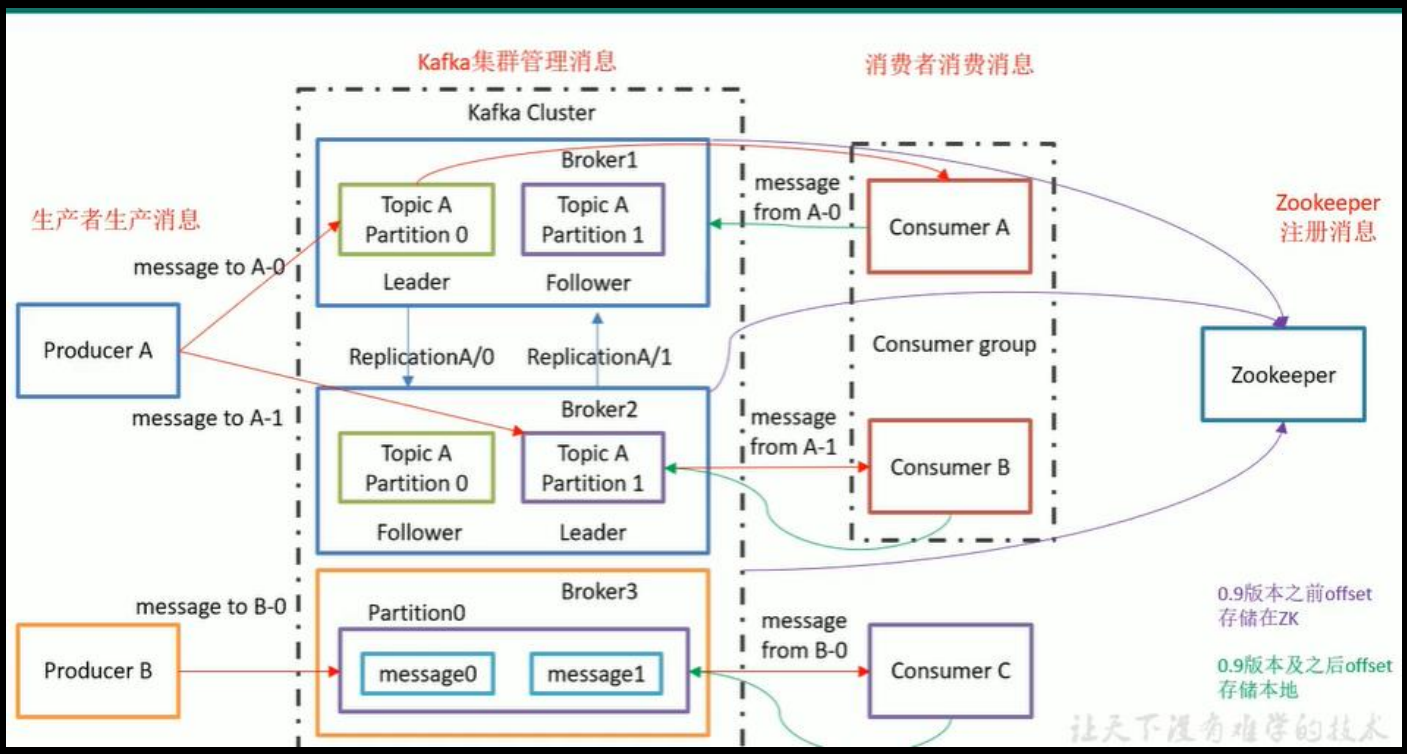


这里面也分为两种：

a) Consumer 自己获取消息：需要不断询问是否有消息

b) 消息队列主动推送信息：consumer 可以性能不一，难以处理

基础架构



注意：partition 和集群不是一个东西，分区是将一个 topic 分为多个区存储，一方面一个区可能存不下，另一方面多个分区有利于负载均衡。

而每个分区都有自己的 replica

1）**Producer**：消息生产者，就是向 kafka broker 发消息的客户端；

2）**Consumer**：消息消费者，向 kafka broker 取消息的客户端；

3）**Consumer Group**（CG）：消费者组，由多个 consumer 组成。**消费者组内每个消费者负责消费不同分区的数据，一个分区只能由一个组内消费者消费；消费者组之间互不影响。**所有的消费者都属于某个消费者组，即**消费者组是逻辑上的一个订阅者。**

4）**Broker**：一台 kafka 服务器就是一个 broker。一个集群由多个 broker 组成。一个 broker 可以容纳多个 topic。

5）**Topic**：可以理解为一个队列，**生产者和消费者面向的都是一个 topic**；

6）**Partition**：为了实现扩展性，一个非常大的 topic 可以分布到多个 broker（即服务器）上，**一个 topic 可以分为多个 partition**，每个 partition 是一个有序的队列；

7）**Replica**：副本，为保证集群中的某个节点发生故障时，该节点上的 partition 数据不丢失，且 kafka 仍然能够继续工作，kafka 提供了副本机制，一个 topic 的每个分区都有若干个副本，一个 **leader** 和若干个 **follower**。

8）**leader**：每个分区多个副本的“主”，生产者发送数据的对象，以及消费者消费数据的对象都是 leader。

9）**follower**：每个分区多个副本中的“从”，实时从 leader 中同步数据，保持和 leader 数据的同步。leader 发生故障时，某个 follower 会成为新的 follower。

Leader 是分区的 leader，而不是 broker 的 leader。

关于消费者组：**一个 topic 的某个分区只能被消费者组里的一个消费者消费**。因为，这一个分区被组内的一个消费者获取后，其他消费者就没必要在访问这个分区了，直接从这个消费者这里获取数据即可。增大了并发性能。

Kafka 快速入门

安装

下载 tar 即可，

```
├──bin
│    └──windows # bat 执行文件，我这里使用的是 windows
├──config # 配置文件
├──kafkakafka01logs // 这应该是根据当前目录生成的数据存储目录
│    ├──course_message-0
│    ├──course_post_message-0
```

```
|   ├──project_message-0
|   ├──test_topic-0 // 这是一个 topic 目录
|   ├──topic01-0
|   ├── __consumer_offsets-0
├──libs
├──logs // 自己新建的日志目录，其中最重要的是 server.log
└──site-docs // 这是文档
```

配置

```
#broker 的全局唯一编号，不能重复
broker.id=0
#删除 topic 功能使能
delete.topic.enable=true
#处理网络请求的线程数量
num.network.threads=3
#用来处理磁盘 IO 的现成数量
num.io.threads=8
#发送套接字的缓冲区大小
socket.send.buffer.bytes=102400
#接收套接字的缓冲区大小
socket.receive.buffer.bytes=102400
#请求套接字的缓冲区大小
socket.request.max.bytes=104857600
#kafka 运行日志存放的路径
log.dirs=/opt/module/kafka/logs
#topic 在当前 broker 上的分区个数
num.partitions=1
#用来恢复和清理 data 下数据的线程数量
num.recovery.threads.per.data.dir=1
#segment 文件保留的最长时间，超时将被删除
log.retention.hours=168
#配置连接 Zookeeper 集群地址
zookeeper.connect=hadoop102:2181,hadoop103:2181,hadoop104:2181
```

```
配置环境变量
[atguigu@hadoop102 module]$ sudo vi /etc/profile
#KAFKA_HOME
export KAFKA_HOME=/opt/module/kafka
export PATH=$PATH:$KAFKA_HOME/bin
[atguigu@hadoop102 module]$ source /etc/profile
```

```
n/kafka-server-start.sh -daemon /opt/module/kafka/config/server.properties"
```

以守护进程形式运行，不是前台进程

Kafka 命令行操作

查看当前服务器中的所有 topic

```
bin/kafka-topics.sh -zookeeper hadoop102:2181 --list
```

创建 topic
bin/kafka-topics.sh -zookeeper hadoop102:2181 --create --replication-factor 3 --partitions 1 --topic first
选项说明:
--topic 定义 topic 名
--replication-factor 定义副本数
--partitions 定义分区数

删除 topic:
bin/kafka-topics.sh -zookeeper hadoop102:2181 --delete --topic first
需要 server.properties 中设置 delete.topic.enable=true 否则只是标记删除。

发送消息:
bin/kafka-console-producer.sh --brokerlist hadoop102:9092 --topic first

消费消息:
bin/kafka-console-consumer.sh --zookeeper hadoop102:2181 --topic first

查看某个 Topic 的详情:
bin/kafka-topics.sh -zookeeper hadoop102:2181 **--describe** --topic first

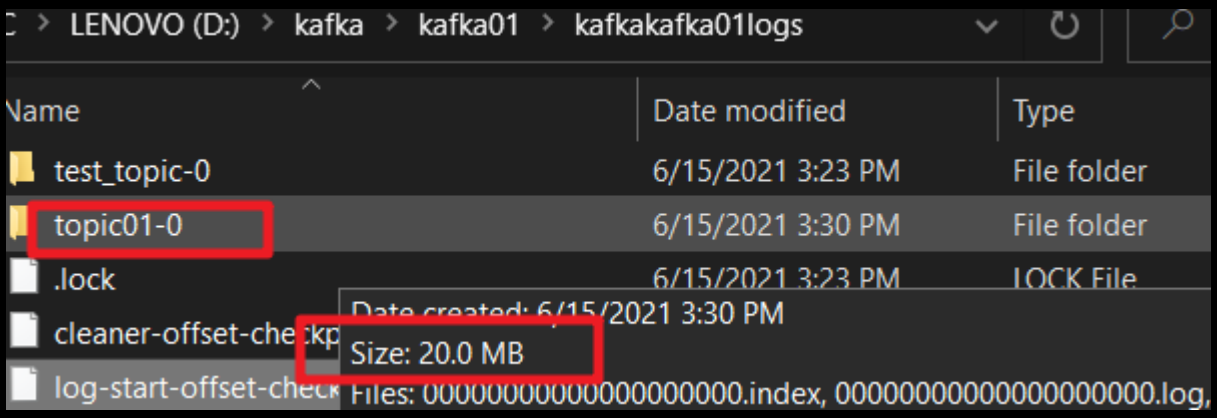
修改分区数:
bin/kafka-topics.sh -zookeeper hadoop102:2181 --alter --topic first --partitions 6

自我探索

首先新建一个 topic:

```
kafka-topics.bat -zookeeper localhost:2181 --create --replication-factor 1 -partitions 1 --topic topic01
```

然后就在数据目录下生成了相应的文件夹:



目录 20MB，这是段大小吗？ 但是我的段大小配置为 100MB

删除

```
D:\kafka\kafka01\bin\windows>kafka-topics.bat --delete -zookeeper localhost:2181 --de
lete --topic test_topic
Topic test_topic is marked for deletion.
Note: This will have no impact if delete.topic.enable is not set to true.
```

```
kafka-topics.bat    --zookeeper localhost:2181 --list
__consumer_offsets
course_message
course_post_message
project_message
topic01 - marked for deletion
```

```
kafka-topics.bat    --zookeeper localhost:2181 --describe
Topic: __consumer_offsets           PartitionCount: 50          ReplicationFactor: 1       Configs:
compression.type=producer,cleanup.policy=compact,segment.bytes=104857600
// 为什么__consumer_offsets 会成为 Topic
            Topic: __consumer_offsets           Partition: 49      Leader: 0           Replicas: 0    Isr: 0
Topic: course_message      PartitionCount: 1           ReplicationFactor: 1       Configs:
            Topic: course_message      Partition: 0       Leader: 0           Replicas: 0        Isr: 0
Topic: course_post_message          PartitionCount: 1           ReplicationFactor: 1       Configs:
            Topic: course_post_message          Partition: 0       Leader: 0           Replicas: 0    Isr: 0
Topic: project_message     PartitionCount: 1           ReplicationFactor: 1       Configs:
            Topic: project_message     Partition: 0       Leader: 0           Replicas: 0        Isr: 0
Topic: topic01     PartitionCount: 1           ReplicationFactor: 1       Configs:            MarkedForDeletion: true
            Topic: topic01     Partition: 0       Leader: none        Replicas: 0        Isr: 0    MarkedForDeletion: true
```

```
kafka-console-producer.bat --bootstrap-server localhost:9092 -topic topic01
>hi
```

我把 topic 全部删除了，然后调用生产者，是会自动创建一个 topic 的？

注意命令格式已经变了，--bootstrap-server 表示指定的 broker

```
kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic topic01
```

一旦调用消费者，马上会生成__consumer_offsets 的 50 个分区

调用这个命令没有任何输出？阻塞而已？

修改分区：

```
kafka-topics.bat --zookeeper localhost:2181 --alter --topic topic01 --partitions 3
WARNING: If partitions are increased for a topic that has a key, the partition logic or ordering of the messages will be
affected
Adding partitions succeeded!
```

如何清空数据：

最重要的是把 zookeeper 中的数据清空掉

```
[zk: localhost:2181(CONNECTED) 0] ls /
[cluster, controller_epoch, brokers, zookeeper, admin, isr_change_notification, consumers, lates
t_producer_id_block, config]
```

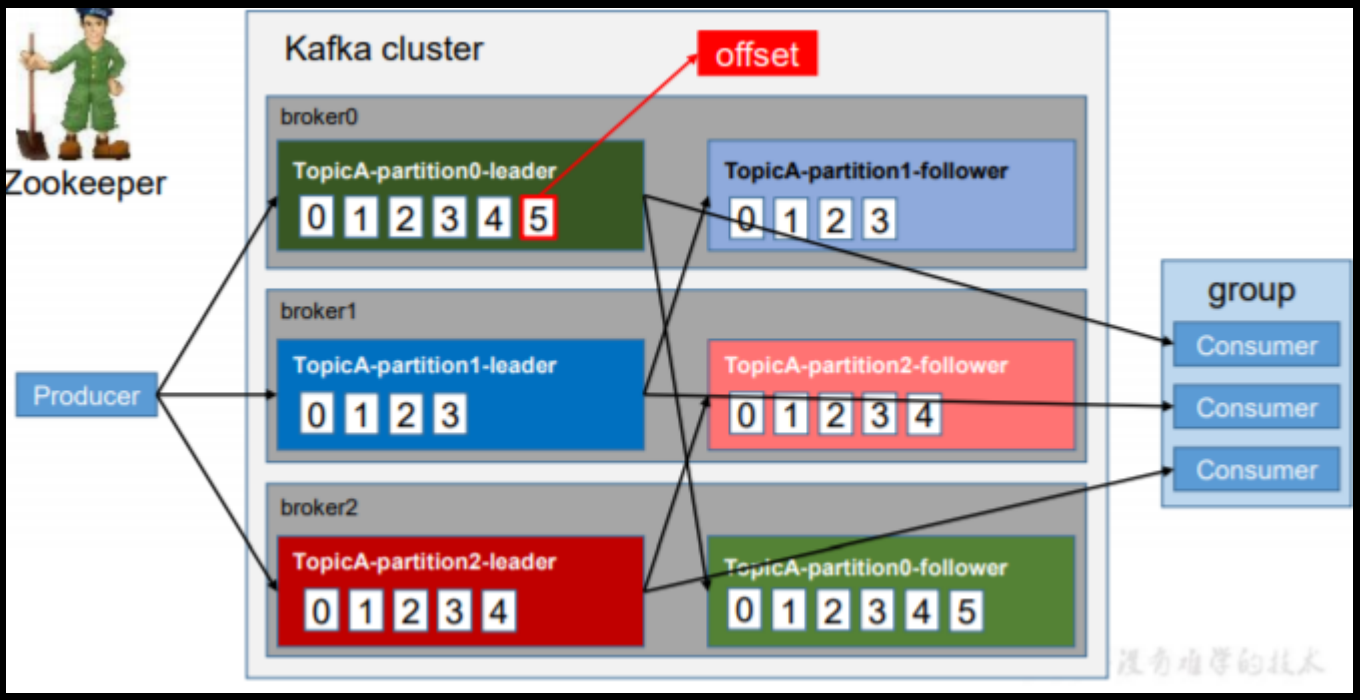
可以使用 deleteall 删除一个节点及其子节点，如果嫌麻烦，可以把 zkdata 下的 version-2 删除掉，相当于重启 ZK

然后 kafka 下面产生的数据可以直接删除

Edwin Xu

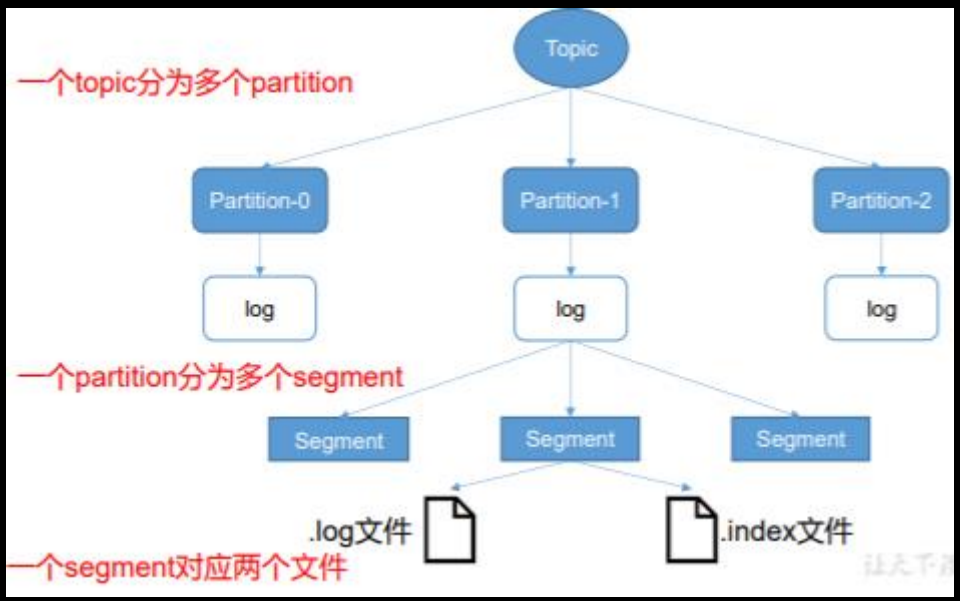
Kafka 架构深入

工作流程和存储结构



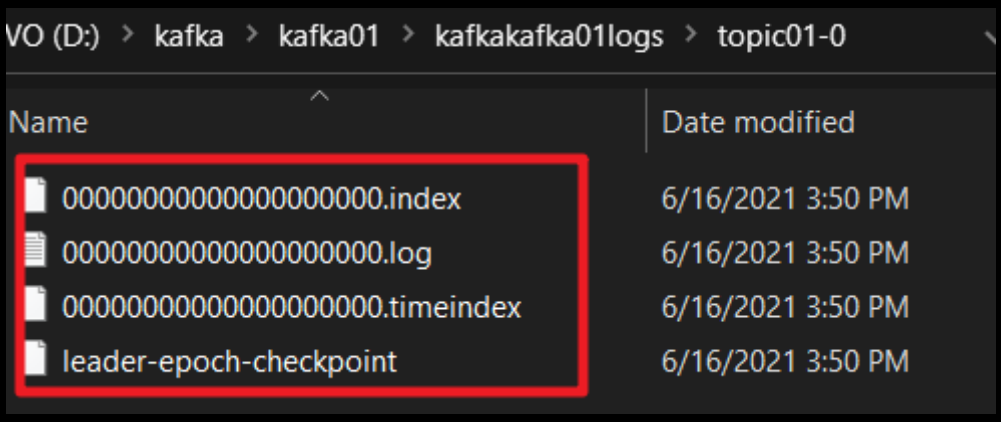
对于 Topic，不同分区不保证有序性，即全局上不保证有序，单一分区是保证有序的

topic 是逻辑上的概念，而 partition 是物理上的概念



每个 partition 对应于一个 log 文件，该 log 文件中存储的就是 producer 生产的数据。Producer 生产的数据会被不断追加到该 log 文件末端，且每条数据都有自己的 offset。消费者组中的每个消费者，都会实时记录自己消费到了哪个 offset，以便出错恢复时，从上次的位置继续

消费。

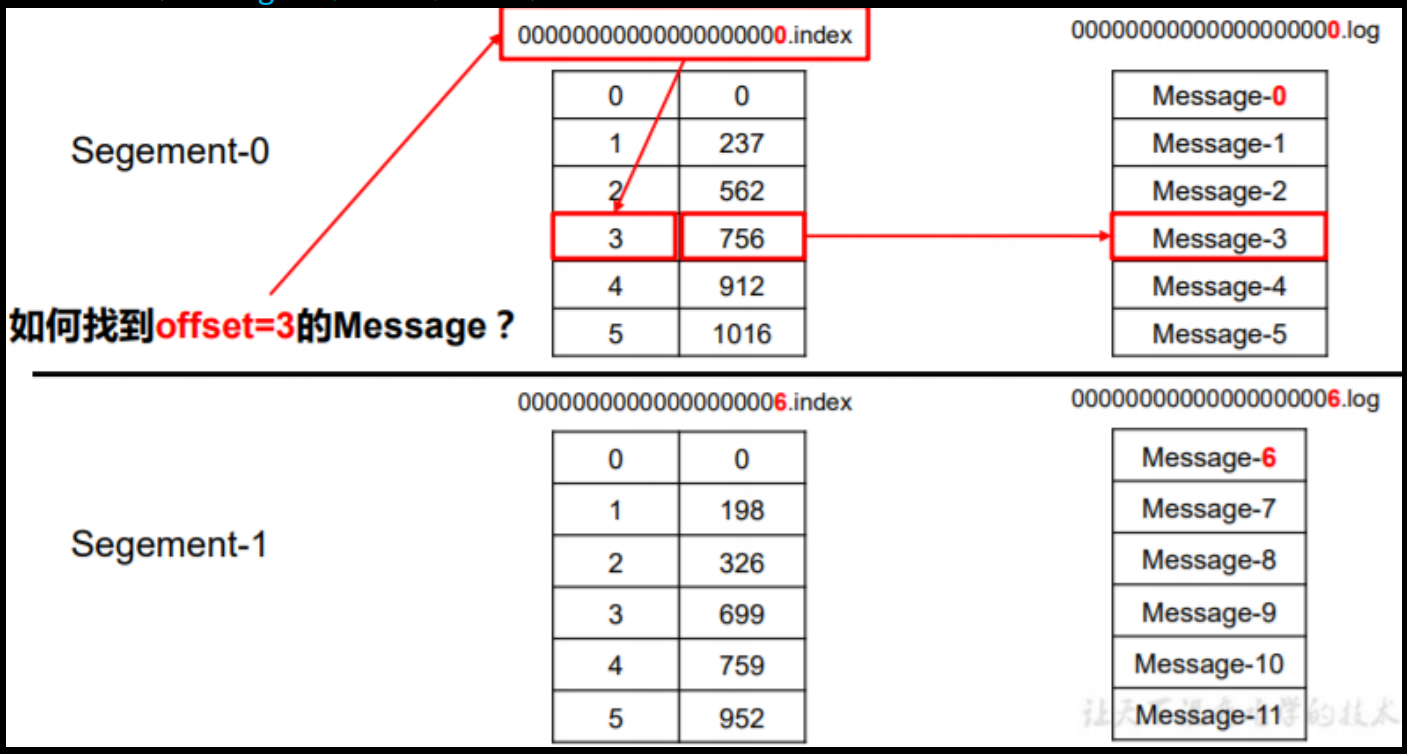


其中.log 就存储的是生产者生产的数据，log.segment.bytes=102400 定义了 log 的最大的段大小，如果数据超过这个大小，就会新建一个文件存储，

由于生产者生产的消息会不断追加到 log 文件末尾，为防止 log 文件过大导致数据定位效率低下，Kafka 采取了分片和索引机制，将每个 partition 分为多个 segment。每个 segment 对应两个文件——“.index”文件和“.log”文件。这些文件位于一个文件夹下，该文件夹的命名规则为：topic 名称+分区序号。

index 和 log 文件以当前 segment 的第一条消息的 offset 命名。
.index、.log 的名称即其中数据的最小偏移量，如上是 0，表示其中第一条数据偏移量是 0.

index 文件和 log 文件的结构示意图：



给定一个 offset，需要先找到是在哪个 segment，这是通过二分查找实现的

然后如何快速定位到该 offset 指向的数据？首先查找 index 文件，其中的 offset 对应的是消息在 log 中的实际位置(物理偏移)，这样就可以快速定位到数据位置。

Edwin Xu

“.index”文件存储大量的索引信息，“.log”文件存储大量的数据，索引文件中的元数据指向对应数据文件中 message 的物理偏移地址。

生产者-分区策略

分区的原因：
1. 方便在集群中扩展
2. 提高并发

分区原则：
生产者生产的数据写到那个分区中呢？

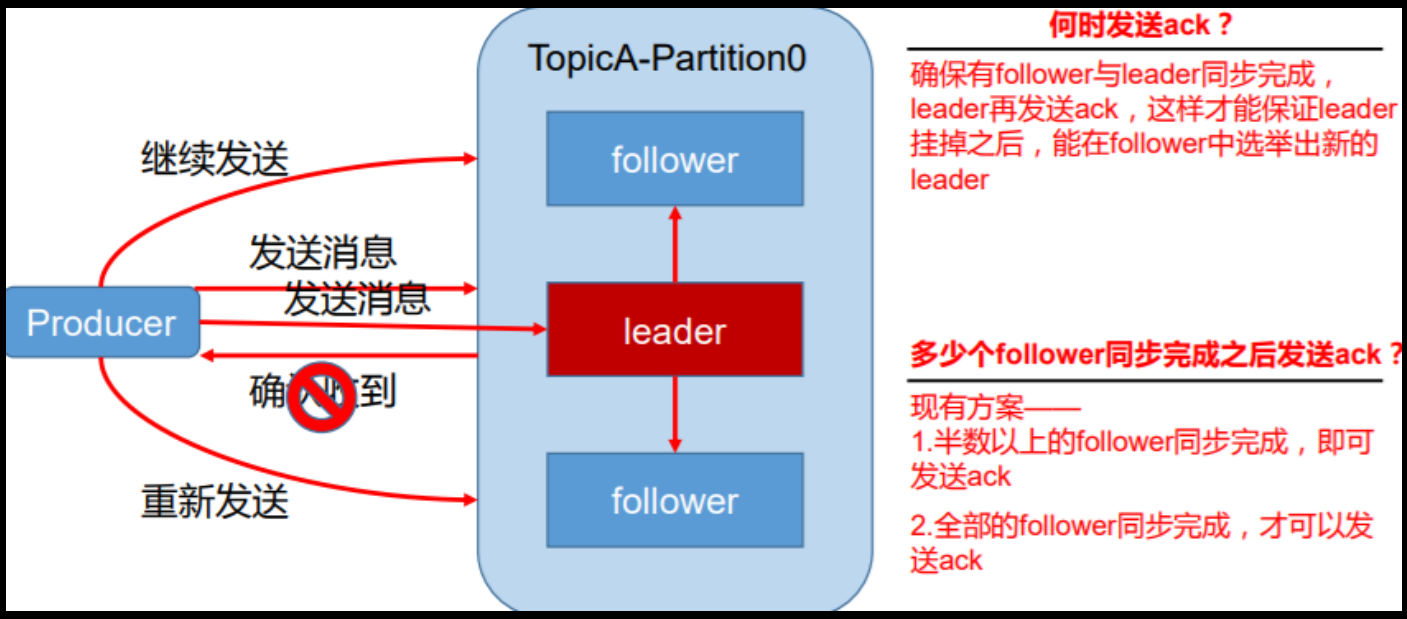
数据会被封装为 ProducerRecord 对象：

```
ProducerRecord(@NotNull String topic, Integer partition, Long timestamp, String key, String value, @Nullable Iterable<Header> headers)
ProducerRecord(@NotNull String topic, Integer partition, Long timestamp, String key, String value)
ProducerRecord(@NotNull String topic, Integer partition, String key, String value, @Nullable Iterable<Header> headers)
ProducerRecord(@NotNull String topic, Integer partition, String key, String value)
ProducerRecord(@NotNull String topic, String key, String value)
ProducerRecord(@NotNull String topic, String value)
```

（1） 指明 partition 的情况下，直接将指明的值直接作为 partiton 值；
（2） 没有指明 partition 值但有 key 的情况下，将 key 的 hash 值与 topic 的 partition 数进行取余得到 partition 值；
（3） 既没有 partition 值又没有 key 值的情况下，第一次调用时随机生成一个整数（后面每次调用在这个整数上自增），将这个值与 topic 可用的 partition 总数取余得到 partition 值，也就是常说的 **round-robin** 算法。

生产者-数据可靠性

为保证 producer 发送的数据，能可靠的发送到指定的 topic，topic 的每个 partition 收到 producer 发送的数据后，都需要向 producer 发送 ack（acknowledgement 确认收到），如果 producer 收到 ack，就会进行下一轮的发送，否则重新发送数据。



副本同步策略：

方案	优点	缺点
半数以上完成同步，就发送 ack	延迟低	选举新的 leader 时，容忍 n 台节点的故障，需要 2n+1 个副本
全部完成同步，才发送 ack	选举新的 leader 时，容忍 n 台节点的故障，需要 n+1 个副本	延迟高

Kafka 选择了第二种方案，原因如下：

1. 同样为了容忍 n 台节点的故障，第一种方案需要 2n+1 个副本，而第二种方案只需要 n+1 个副本，而 Kafka 的每个分区都有大量的数据，第一种方案会造成大量数据的冗余。
2. 虽然第二种方案的网络延迟会比较高，但网络延迟对 Kafka 的影响较小。

ISR

采用第二种方案之后，设想以下情景：leader 收到数据，所有 follower 都开始同步数据，但有一个 follower，因为某种故障，迟迟不能与 leader 进行同步，那 leader 就要一直等下去，直到它完成同步，才能发送 ack。这个问题怎么解决呢？

Leader 维护了一个动态的 **in-sync replica set (ISR)**，意为**和 leader 保持同步的 follower 集合**。当 ISR 中的 follower 完成数据的同步之后，leader 就会给 follower 发送 ack。如果 follower 长时间未向 leader 同步数据，则该 follower 将被踢出 ISR，该时间阈值由 replica.lag.time.max.ms 参数设定。Leader 发生故障之后，就会从 ISR 中选举新的 leader。

Edwin Xu

```
[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --describe --topic first --zookeeper hadoop102:21
81
.Topic:first    PartitionCount:2        ReplicationFactor:2     Configs:
        Topic: first    Partition: 0    Leader: 2       Replicas: 2,0   Isr: 2,0
        Topic: first    Partition: 1    Leader: 0       Replicas: 0,1   Isr: 1,0
```

ack 应答机制

以 Kafka 为用户提供了三种可靠性级别，用户根据对可靠性和延迟的要求进行权衡：

1. 0：producer 不等待 broker 的 ack，这一操作提供了一个最低的延迟，broker 一接收到还没有写入磁盘就已经返回，当 broker 故障时有可能丢失数据；
2. 1：producer 等待 broker 的 ack，partition 的 leader 落盘成功后返回 ack，如果在 follower 同步成功之前 leader 故障，那么将会丢失数据；
3. -1（all）：producer 等待 broker 的 ack，partition 的 leader 和 follower 全部落盘成功后才返回 ack。但是如果在 follower 同步完成后，broker 发送 ack 之前，leader 发生故障，那么会造成**数据重复**。

故障处理细节——一致性

LEO：指的是每个副本最大的 offset；

HW：指的是**消费者能见到的最大的 offset**，ISR 队列中最小的 LEO

1. follower 故障：follower 发生故障后会被临时踢出 ISR，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。等该 follower 的 LEO 大于等于该 Partition 的 HW，即 follower 追上 leader 之后，就可以重新加入 ISR 了。

2. **leader 故障**：leader 发生故障之后，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。

Exactly Once 语义

将服务器的 ACK 级别设置为-1，可以保证 Producer 到 Server 之间不会丢失数据，即 **At Least Once** 语义。相对的，将服务器 ACK 级别设置为 0，可以保证生产者每条消息只会被发送一次，即 **At Most Once** 语义。

At Least Once 可以保证数据不丢失，但是不能保证数据不重复；相对的，At Most Once 可以保证数据不重复，但是不能保证数据不丢失。但是，对于一些非常重要的信息，比如说交易数据，下游数据消费者要求数据既不重复也不丢失，即 **Exactly Once** 语义

0.11 版本的 Kafka，引入了一项重大特性：幂等性。所谓的幂等性就是指 Producer 不论向 Server 发送多少次重复数据，Server 端都只会持久化一条。幂等性结合 At Least Once 语义，就构成了 Kafka 的 Exactly Once 语义。

At Least Once + 幂等性 = Exactly Once

要启用幂等性，只需要将 Producer 的参数中 **enable.idompotence** 设置为 true 即可(开启后分区策略默认为-1)。Kafka 的幂等性实现其实就是将原来下游需要做的去重放在了数据上游。开启幂等性的 **Producer 在初始化的时候会被分配一个 PID，发往同一 Partition 的消息会附带 Sequence Number。而 Broker 端会对<PID，Partition，SeqNumber>做缓存**，当具有相同主键的消息提交时，Broker 只会持久化一条。

但是 PID 重启就会变化，同时不同的 Partition 也具有不同主键，所以幂等性无法保证跨分区跨会话的 Exactly Once。

消费者-消费方式

consumer 采用 pull（拉）模式从 broker 中读取数据

pull 模式不足之处是，如果 kafka 没有数据，消费者可能会陷入循环中，一直返回空数据。针对这一点,Kafka 的消费者在消费数据时会传入一个时长参数 timeout,如果当前没有数据可供消费，consumer 会等待一段时间之后再返回，这段时长即为 timeout。

消费者-分区分配策略

一个 consumer group 中有多个 consumer，一个 topic 有多个 partition，所以必然会涉及到 partition 的分配问题，即确定那个 partition 由哪个 consumer 来消费。

Kafka 有 2，种分配策略，最新还有一个 StickyAssignor 策略。

将分区的所有权从一个消费者移到另一个消费者称为**重新平衡（rebalance）**。当以下事件发生时，Kafka 将会进行一次分区分配：

1. **同一个 Consumer Group 内新增消费者**
2. **消费者离开当前所属的 Consumer Group，包括 shuts down 或 crashes**
3. **订阅的主题新增分区**

目前我们还不能自定义分区分配策略，只能通过 **partition.assignment.strategy** 参数选择 range 或 roundrobin。**默认的值是 range**。

Range 对应的类是 org.apache.kafka.clients.consumer.RangeAssignor，此外 RoundRobinAssignor 和 StickyAssignor

消费者客户端参数 partition.asssignment.strategy 可以配置多个分配策略，彼此之间以逗号分隔。？？？ 是按先后顺序使用吧，前面不行才是后面一个？？？

1. Range:
   Range 是对 Topic 而言的，首先对同一个 topic 里面的分区按照序号进行排序，并对消费者按照字母顺序进行排序。
   然后用 Partitions 分区的个数除以消费者线程的总数来决定每个消费者线程消费几个分区。如果除不尽，那么前面几个消费者线程将会多消费一个分区。

   假设 n=分区数/消费者数量，m=分区数%消费者数量，那么前 m 个消费者每个分配 n+1 个分区，后面的（消费者数量-m）个消费者每个分配 n 个分区。

   假如有 10 个分区，3 个消费者线程，把分区按照序号排列 0，1，2，3，4，5，6，7，8，9；消费者线程为 C1-0，C2-0，C2-1

   ```
   C1-0：0，1，2，3
   C2-0：4，5，6
   C2-1：7，8，9
   ```

   若两个主题 T1,T2

   ```
   C1-0：T1（0，1，2，3） T2（0，1，2，3）
   C2-0：T1（4，5，6） T2（4，5，6）
   C2-1：T1（7，8，9） T2（7，8，9）
   ```

   如果 topic 过多，会导致前面的消费者消费过多分区，这是 range 的弊端

2. RoundRobin:
   将消费组内所有消费者以及消费者所订阅的所有 topic 的 partition 按照字典序排序，然后通

过轮询方式逐个将分区以此分配给每个消费者。

使用 RoundRobin 策略有两个前提条件必须满足：
a.同一个消费者组里面的所有消费者的 num.streams（**消费者消费线程数**）必须相等；
b.每个消费者订阅的主题必须相同。

Num.streams：启动参数，消费者的个数

RoundRobin 策略的工作原理：将所有主题的分区组成 TopicAndPartition 列表，然后对 TopicAndPartition 列表按照 hashCode 进行排序

假如按照 hashCode 排序完的 topic-partitions 组依次为 T1-5，T1-3，T1-0，T1-8，T1-2，T1-1，T1-4，T1-7，T1-6，T1-9，我们的消费者线程排序为 C1-0，C1-1，C2-0，C2-1，最后分区分配的结果为：

C1-0 将消费 T1-5, T1-2, T1-6 分区；
C1-1 将消费 T1-3, T1-1, T1-9 分区；
C2-0 将消费 T1-0, T1-4 分区；
C2-1 将消费 T1-8, T1-7 分区；

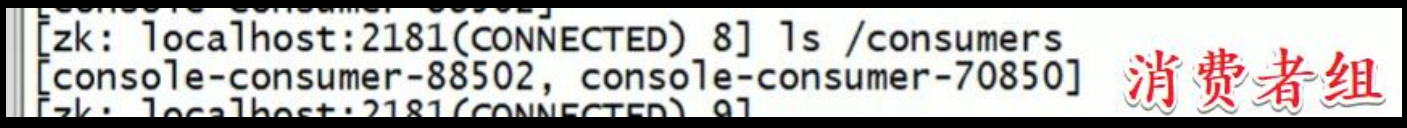
3. StickyAssignor
我们再来看一下 StickyAssignor 策略，“sticky”这个单词可以翻译为“粘性的”，Kafka 从 0.11.x 版本开始引入这种分配策略，它主要有两个目的：

分区的分配要尽可能的均匀，分配给消费者者的主题分区数最多相差一个；
分区的分配尽可能的与上次分配的保持相同。
当两者发生冲突时，第一个目标优先于第二个目标。鉴于这两个目标，StickyAssignor 策略的具体实现要比 RangeAssignor 和 RoundRobinAssignor 这两种分配策略要复杂很多。

Offset 的维护

Consumer 需要实时记录自己消费到那个消息了，以便宕机后迅速恢复

Kafka0.9 版本之前，offset 是保存在 ZK 中的：



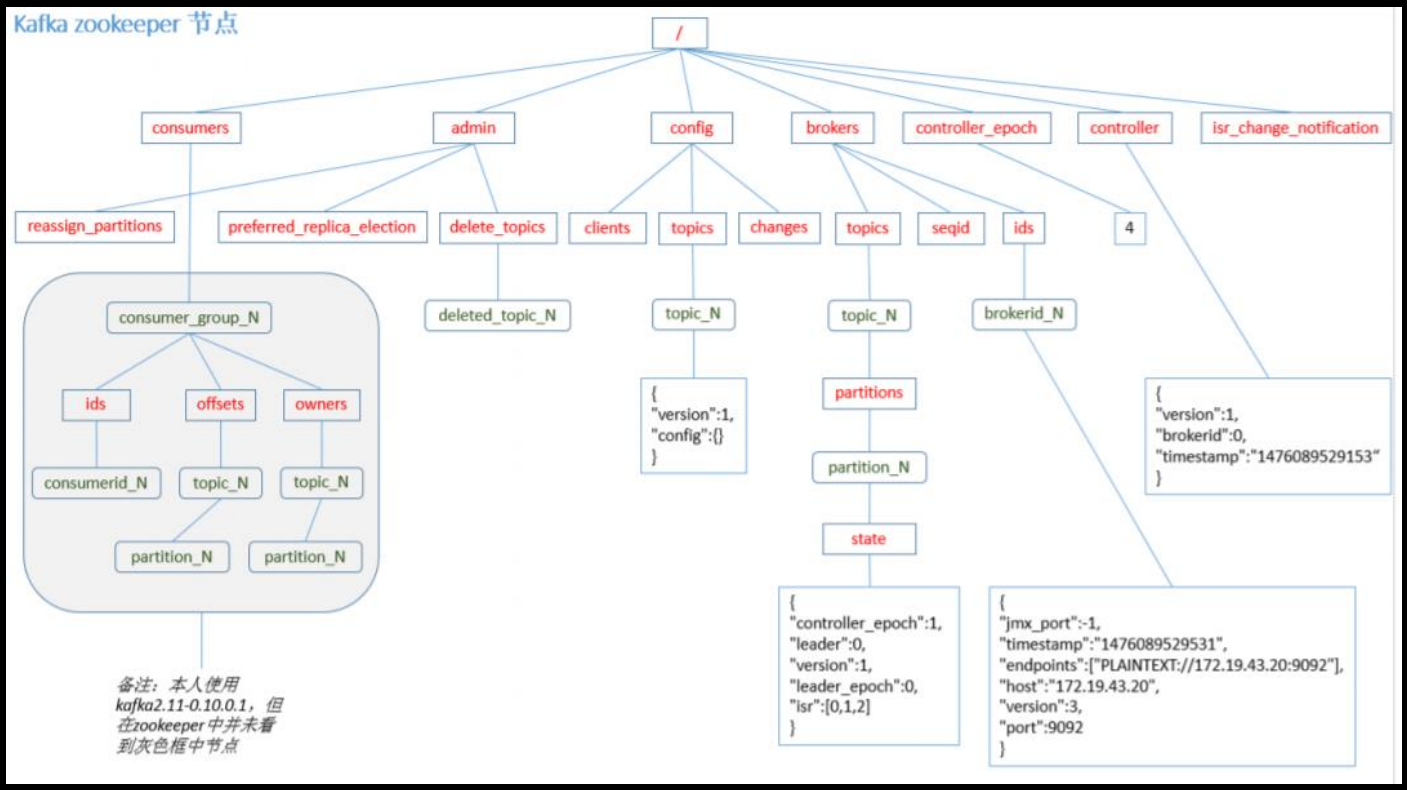
```
[zk: localhost:2181(CONNECTED) 8] ls /consumers
[console-consumer-88502, console-consumer-70850]
[zk: localhost:2181(CONNECTED) 9] ls /consumers/console-consumer-88502
[ids, owners, offsets]
[zk: localhost:2181(CONNECTED) 10] ls /consumers/console-consumer-88502/o

owners    offsets
[zk: localhost:2181(CONNECTED) 10] ls /consumers/console-consumer-88502/offsets
[bigdata]
[zk: localhost:2181(CONNECTED) 11] ls /consumers/console-consumer-88502/offsets/bigdata
[0, 1]
[zk: localhost:2181(CONNECTED) 12] ls /consumers/console-consumer-88502/offsets/bigdata/0
[]
[zk: localhost:2181(CONNECTED) 13] ls /consumers/console-consumer-88502/offsets/bigdata/0
```

消费者组

offset

topic

分区



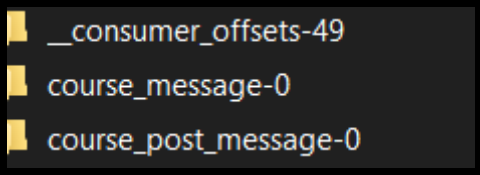
从 0.9 版本开始，consumer 默认将 offset 保存在 Kafka 一个内置的 topic 中，该 topic 为**__consumer_offsets**。

修改配置文件 consumer.properties

```
exclude.internal.topics=false # 让普通的消费者消费
```

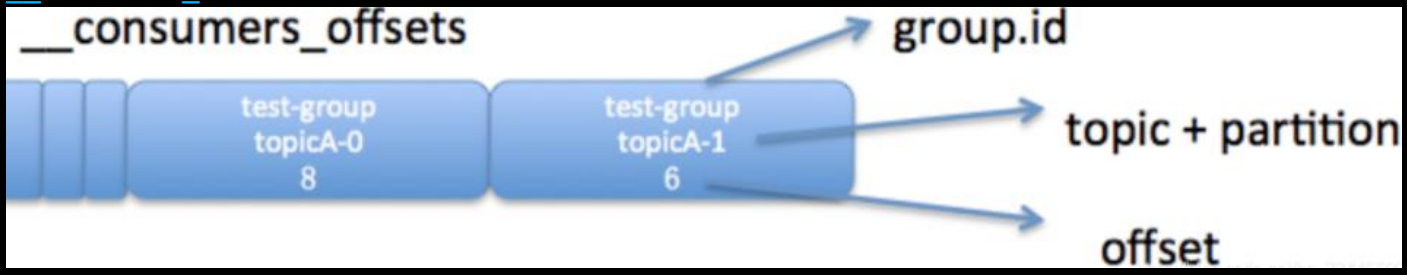
__consumer_offsets

kafka 在 0.10.x 版本后**默认将消费者组的位移提交到自带的 topic __consumer_offsets 里面**，当有消费者第一次消费 kafka 数据时就会自动创建，它的副本数不受集群配置的 topic 副本数限制，分区数默认 50（可以配置），默认压缩策略为 compact



保存 consumer 提交的位移。

__consumer_offsets 的每条消息格式大致如图所示：



可以想象成一个 KV 格式的消息，**key 就是一个三元组：group.id+topic+分区号，而 value 就是 offset 的值。**

考虑到一个 kafka 生成环境中可能有很多 consumer 和 consumer group，如果这些 consumer 同时提交位移，则必将加重 __consumer_offsets 的写入负载，因此 kafka 默认为该 topic 创建了 50 个分区，并且对每个 group.id 做哈希求模运算，从而将负载分散到不同的 __consumer_offsets 分区上。

一般情况下，当集群中第一次有消费者消费消息时会自动创建 __consumer_offsets，它的**副本因子受 offsets.topic.replication.factor 参数的约束，默认值为 3**（注意：该参数的使用限制在 0.11.0.0 版本发生变化），分区数可以通过 **offsets.topic.num.partitions** 参数设置，默认值为 50。

```
# consumer_offsets 的副本数，默认是 3
offsets.topic.replication.factor=1
# consumer_offsets 的分区数， 默认是 50
offsets.topic.num.partitions=2
```

https://blog.csdn.net/qq_33446500/article/details/105890655

Kafka 高效读写数据

顺序写磁盘

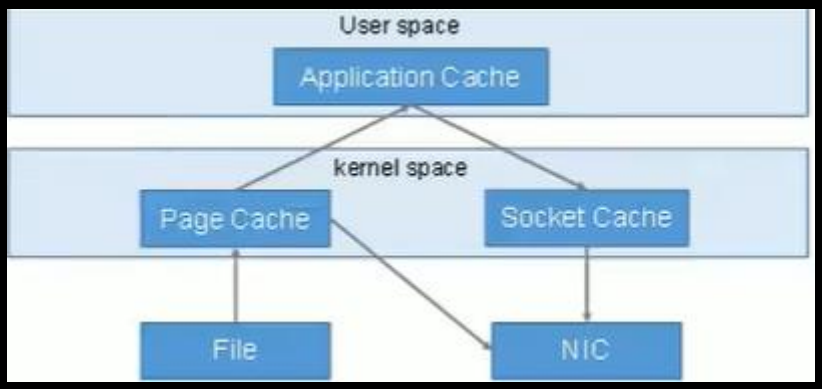
Producer 追加到 log 文件，是顺序写，顺序写速率很高，因为节省了磁头寻址的时间。

零拷贝

Edwin Xu

写到 OS 笔记中

1. Mmap + write
2. Sendfile



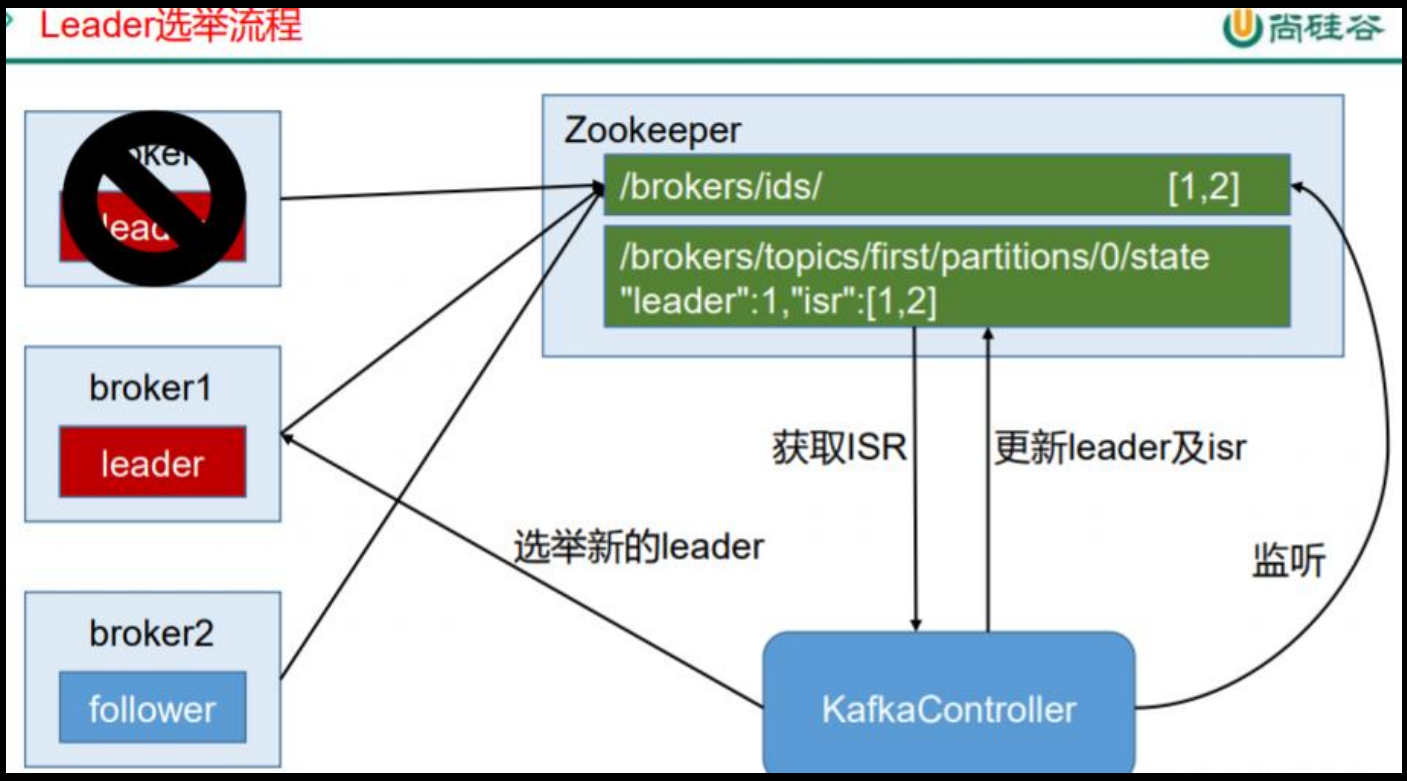
NIC:网卡

Zookeeper 在 Kafka 中的作用

Kafka 集群中有一个 broker 会被选举为 Controller，负责管理集群 broker 的上下线，所有 topic 的分区副本分配和 leader 选举等工作。

Controller 的管理工作都是依赖于 Zookeeper 的。

partition 的 leader 选举过程:



Kafka 事务

Kafka 从 0.11 版本开始引入了事务支持。事务可以保证 Kafka 在 Exactly Once 语义的基础上，**生产和消费可以跨分区和会话**，要么全部成功，要么全部失败。

Producer 事务

为了实现跨分区跨会话的事务，需要引入一个全局唯一的 Transaction ID，并将 Producer 获得的 PID 和 Transaction ID 绑定。这样当 Producer 重启后就可以通过正在进行的 Transaction
ID 获得原来的 PID。
为了管理 Transaction，Kafka 引入了一个新的组件 Transaction Coordinator。Producer 就是通过和 Transaction Coordinator 交互获得 Transaction ID 对应的任务状态。Transaction
Coordinator 还负责将事务所有写入 Kafka 的一个内部 Topic，这样即使整个服务重启，由于事务状态得到保存，进行中的事务状态可以得到恢复，从而继续进行。

Consumer 事务

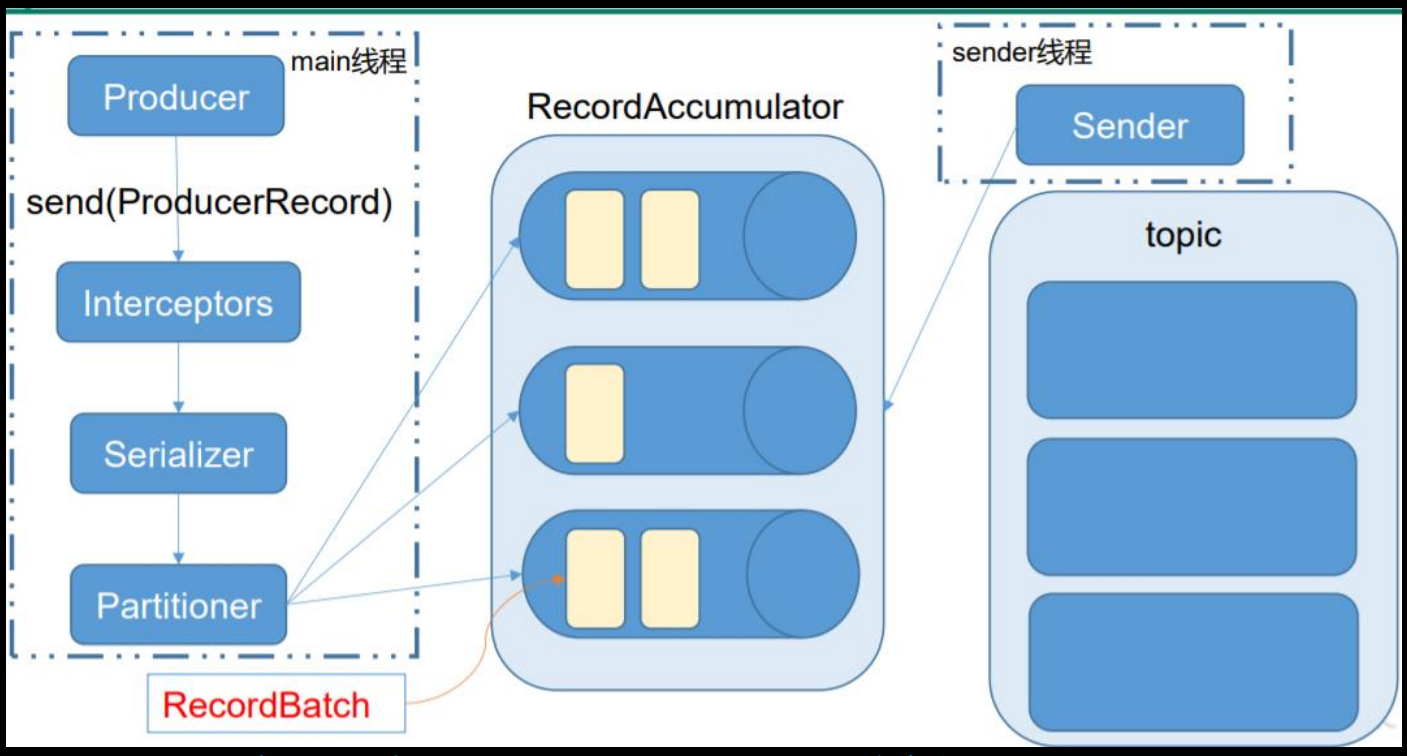
上述事务机制主要是从 Producer 方面考虑，对于 Consumer 而言，事务的保证就会相对较弱，尤其时无法保证 Commit 的信息被精确消费。这是由于 Consumer 可以通过 offset 访问任意信息，而且不同的 Segment File 生命周期不同，同一事务的消息可能会出现重启后被删除的情况。

Kafka API

Producer API

消息发送流程

Kafka 的 Producer 发送消息采用的是**异步发送**的方式。在消息发送的过程中，涉及到了两个线程——**main 线程和 Sender 线程**，以及一个线程共享变量——**RecordAccumulator**。main 线程将消息发送给 RecordAccumulator，Sender 线程不断从 RecordAccumulator 中拉取消息发送到 Kafka broker。



batch.size：只有数据积累到 batch.size 之后，sender 才会发送数据。

linger.ms：如果数据迟迟未达到 batch.size，sender 等待 linger.time 之后就会发送数据。

异步发送 API

```
<dependency>
<groupId>org.apache.kafka</groupId>
<artifactId>kafka-clients</artifactId>
<version>0.11.0.0</version>
</dependency>
```

1. **KafkaProducer**：需要创建一个生产者对象，用来发送数据
2. **ProducerConfig**：获取所需的一系列配置参数
3. **ProducerRecord**：**每条数据都要封装成一个 ProducerRecord 对象**

```
Properties props = new Properties();
 //kafka 集群，broker-list
 props.put("bootstrap.servers", "hadoop102:9092");
 props.put("acks", "all");
 //重试次数
 props.put("retries", 1);
 //批次大小
 props.put("batch.size", 16384);
 //等待时间
 props.put("linger.ms", 1);
 //RecordAccumulator 缓冲区大小
 props.put("buffer.memory", 33554432);
 props.put("key.serializer","org.apache.kafka.common.serialization.StringSerializer");
 props.put("value.serializer","org.apache.kafka.common.serialization.StringSerializer");
 Producer<String, String> producer = new KafkaProducer<>(props);
 for (int i = 0; i < 100; i++) {
    producer.send(new ProducerRecord<String, String>("first", Integer.toString(i), Integer.toString(i)));
 }
 producer.close();
```

```
}
```

有回调：

```
producer.send(new ProducerRecord<String, String>("first",
Integer.toString(i), Integer.toString(i)), new Callback() {
 //回调函数，该方法会在 Producer 收到 ack 时调用，为异步调用
 @Override
 public void onCompletion(RecordMetadata metadata,
Exception exception) {
 if (exception == null) {
 System.out.println("success->" +
metadata.offset());
 } else {
 exception.printStackTrace();
 }
 }
 });
```

由于是异步的，所以就算是单一分区内，也无法保证绝对的有序性，因为一个消息发送失败，再次尝试发送成功，很可能会滞后于一些本来在后面的消息。

同步发送：
由于 send 方法返回的是一个 Future 对象，根据 Futrue 对象的特点，我们也可以实现同步发送的效果，只需在调用 Future 对象的 get 方发即可

```
producer.send(new ProducerRecord<String, String>("first",
Integer.toString(i), Integer.toString(i))).get();
```

Consumer API

offset 的维护是 Consumer 消费数据是必须考虑的问题。

1. KafkaConsumer：需要创建一个消费者对象，用来消费数据
2. ConsumerConfig：获取所需的一系列配置参数
3. ConsuemrRecord：每条数据都要封装成一个 ConsumerRecord 对象

为了使我们能够专注于自己的业务逻辑，Kafka 提供了自动提交 offset 的功能。自动提交 offset 的相关参数：

1. enable.auto.commit：是否开启自动提交 offset 功能
2. auto.commit.interval.ms：自动提交 offset 的时间间隔

自动提交：

```
public static void main(String[] args) {
 Properties props = new Properties();
 props.put("bootstrap.servers", "hadoop102:9092");
 props.put("group.id", "test");
 props.put("enable.auto.commit", "true");
 props.put("auto.commit.interval.ms", "1000");
```

```
  props.put("key.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
  props.put("value.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
  KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
  consumer.subscribe(Arrays.asList("first"));
  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());
  }
  }
```

Shell 编程

整合 https://blog.csdn.net/yuanlong122716/article/details/105160545

Edwin Xu