

讲师：元敬

Kafka是一款开源的消息引擎系统



1. 消息队列(MQ)

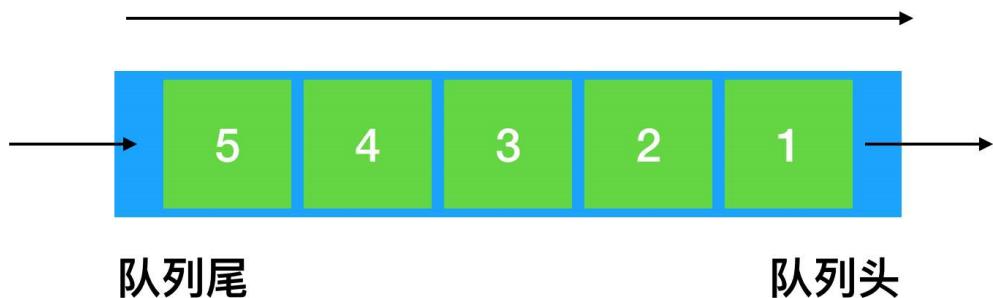
1.1 什么是消息队列

消息队列不知道大家看到这个词的时候，会不会觉得它是一个比较高端的技术。消息队列，一般我们会简称它为MQ (Message Queue) .

消息队列是一种帮助开发人员解决系统间异步通信的中间件，常用于解决系统解耦和请求的削峰平谷的问题。

队列 (Queue) :

Queue 是一种先进先出的数据结构，容器

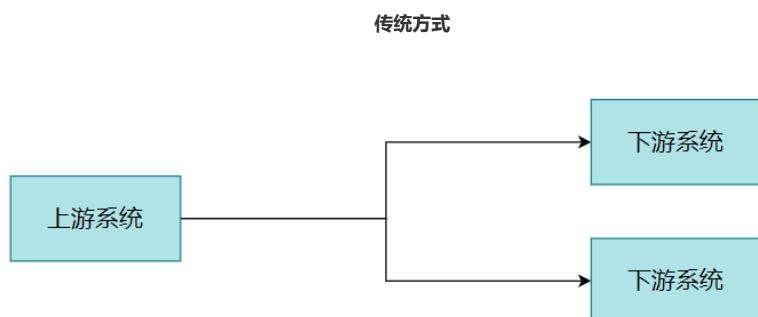


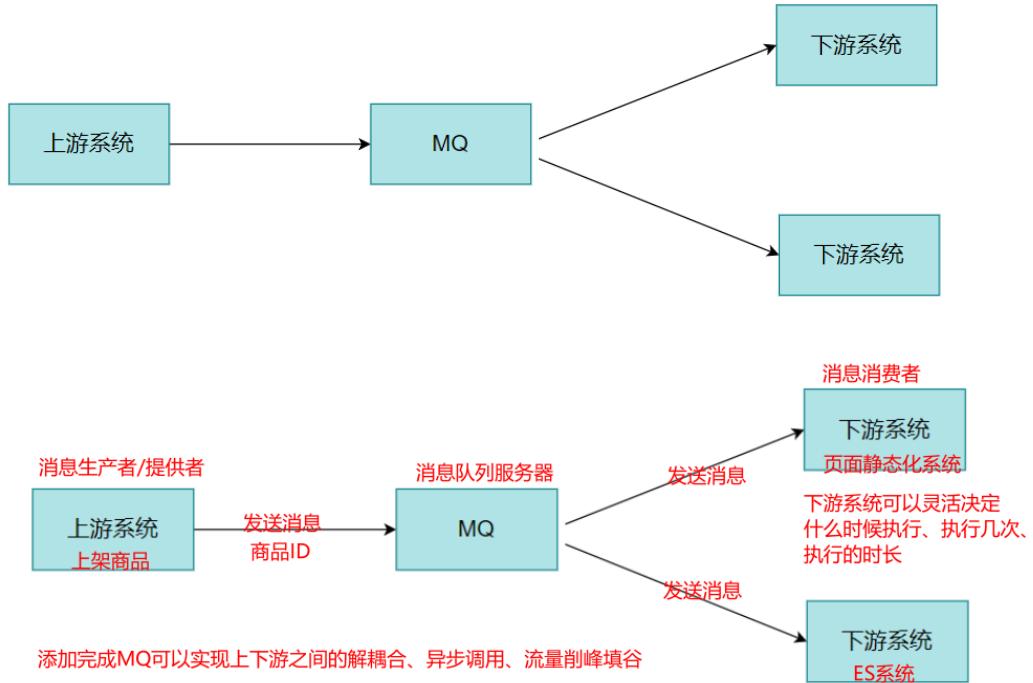
消息 (Message) :

不同应用之间传送的数据。

消息队列:

我们可以把消息队列比作是一个存放消息的容器，当我们需要使用消息的时候可以取出消息供自己使用。消息队列是分布式系统中重要的组件，使用消息队列主要是为了通过异步处理提高系统性能和削峰、降低系统耦合性。队列 Queue 是一种先进先出的数据结构，所以消费消息时也是按照顺序来消费的。比如生产者发送消息1,2,3...对于消费者就会按照1,2,3...的顺序来消费。





1.2 消息队列的应用场景

消息队列在实际应用中包括如下四个场景：

- 1) **应用耦合**: 多应用间通过消息队列对同一消息进行处理，避免调用接口失败导致整个过程失败；
- 2) **异步处理**: 多应用对消息队列中同一消息进行处理，应用间并发处理消息，相比串行处理，减少处理时间；
- 3) **限流削峰**: 广泛应用于秒杀或抢购活动中，避免流量过大导致应用系统挂掉的情况；
- 4) **消息驱动的系统**: 系统分为消息队列、消息生产者、消息消费者，生产者负责产生消息，消费者(可能有多个)负责对消息进行处理

下面详细介绍上述四个场景以及消息队列如何在上述四个场景中使用

1.2.1 异步处理

具体场景：

用户为了使用某个应用，进行注册，系统需要发送注册邮件并验证短信。

对这两个子系统操作的处理方式有两种：串行及并行。

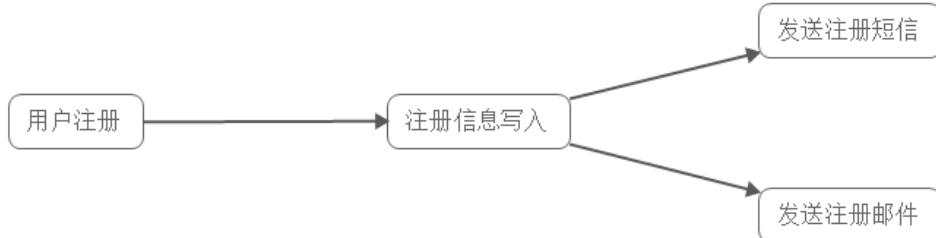
涉及到三个子系统：注册系统、邮件系统、短信系统

- 1) 串行方式：新注册信息生成后，先发送注册邮件，再发送验证短信；



在这种方式下，需要最终发送验证短信后再返回给客户端。

- 2) 并行处理：新注册信息写入后，由发短信和发邮件并行处理



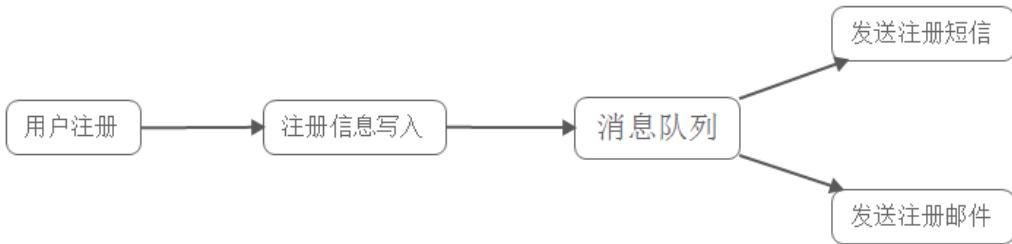
在这种方式下，发短信和发邮件需处理完成后返回给客户端。

假设以上三个子系统处理的时间均为50ms，且不考虑网络延迟，则总的处理时间：

串行： $50+50+50=150\text{ms}$

并行： $50+50 = 100\text{ms}$

如果引入消息队列，在来看整体的执行效率：



在写入消息队列后立即返回成功给客户端，则总的响应时间依赖于写入消息队列的时间，而写入消息队列的时间本身是可以很快的，基本可以忽略不计，因此总的处理时间为50ms，相比串行提高了2倍，相比并行提高了一倍；

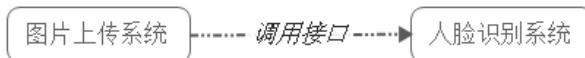
1.2.2 应用耦合

具体场景：

用户使用QQ相册上传一张图片，人脸识别系统会对该图片进行人脸识别。

一般的做法是，服务器接收到图片后，图片上传系统立即调用人脸识别系统，调用完成后再返回成功，如下图所示：

调用方式：webService、Http协议（HttpClient、RestTemplate）、Tcp协议（Dubbo）



该方法有如下缺点：

- 1) 人脸识别系统被调失败，导致图片上传失败；
- 2) 延迟高，需要人脸识别系统处理完成后，再返回给客户端，即使用户并不需要立即知道结果；
- 3) 图片上传系统与人脸识别系统之间互相调用，需要做耦合；

若使用消息队列：



客户端上传图片后，图片上传系统将图片信息批次写入消息队列，直接返回成功；

人脸识别系统则定时从消息队列中取数据，完成对新增图片的识别。

图片上传系统并不需要关心人脸识别系统是否对这些图片信息的处理、以及何时对这些图片信息进行处理。

事实上，由于用户并不需要立即知道人脸识别结果，人脸识别系统可以选择不同的调度策略，按照闲时、忙时、正常时间，对队列中的图片信息进行处理。

1.2.3 限流削峰

具体场景：

购物网站开展秒杀活动，一般由于瞬时访问量过大，服务器接收过大，会导致流量暴增，相关系统无法处理请求甚至崩溃。

而加入消息队列后，系统可以从消息队列中取数据，相当于消息队列做了一次缓冲。



该方法有如下优点：

请求先入消息队列，而不是由业务处理系统直接处理，做了一次缓冲，极大地减少了业务处理系统的压力；

队列长度可以做限制，事实上，秒杀时，后入队列的用户无法秒杀到商品，这些请求可以直接被抛弃，返回活动已结束或商品已售完信息；

1.2.4 消息事件驱动的系统

具体场景：

用户新上传了一批照片 ----> 人脸识别系统需要对这个用户的所有照片进行聚类 -----> 由对账系统重新生成用户的人脸索引（加快查询）。

这三个子系统间由消息队列连接起来，前一个阶段的处理结果放入队列中，后一个阶段从队列中获取消息继续处理。



该方法有如下优点：

避免了直接调用下一个系统导致当前系统失败；

每个子系统对于消息的处理方式可以更为灵活，可以选择收到消息时就处理，可以选择定时处理，也可以划分时间段按不同处理速度处理；

1.3 消息队列的两种模式

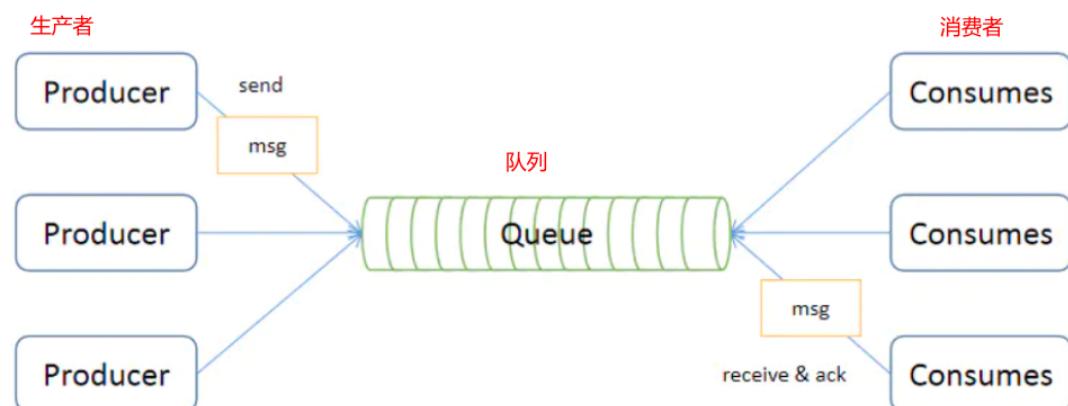
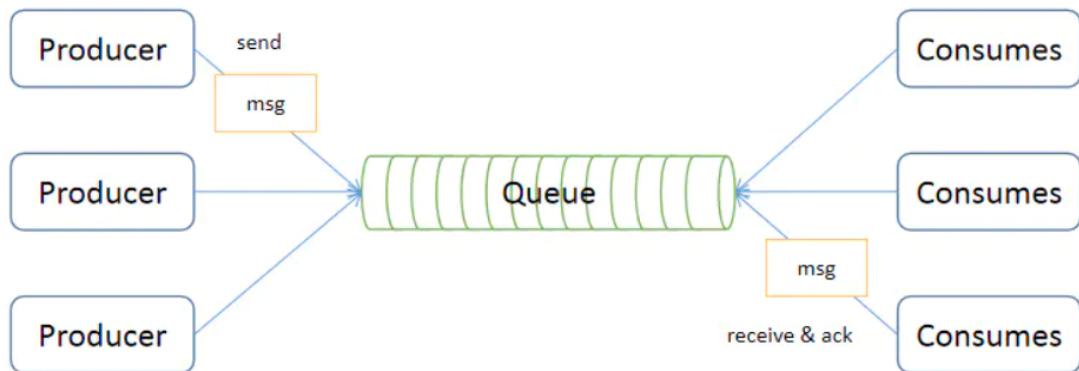
消息队列包括两种模式，**点对点模式** (point to point, queue) 和**发布/订阅模式** (publish/subscribe, topic)

1.3.1 点对点模式

点对点模式下包括三个角色

- 消息队列
- 发送者 (生产者)
- 接收者 (消费者)

每个消息都被发送到一个特定的队列，接收者从队列中获取消息。队列保留着消息，可以放在内存中也可以持久化，直到他们被消费或超时。



点对点模式特点：

- 每个消息只有一个消费者，一旦被消费，消息就不再在消息队列中；
- 发送者和接收者间没有依赖性，发送者发送消息之后，不管有没有接收者在运行，都不会影响到发送者下次发送消息；
- 接收者在成功接收消息之后需向队列应答成功，以便消息队列删除当前接收的消息；

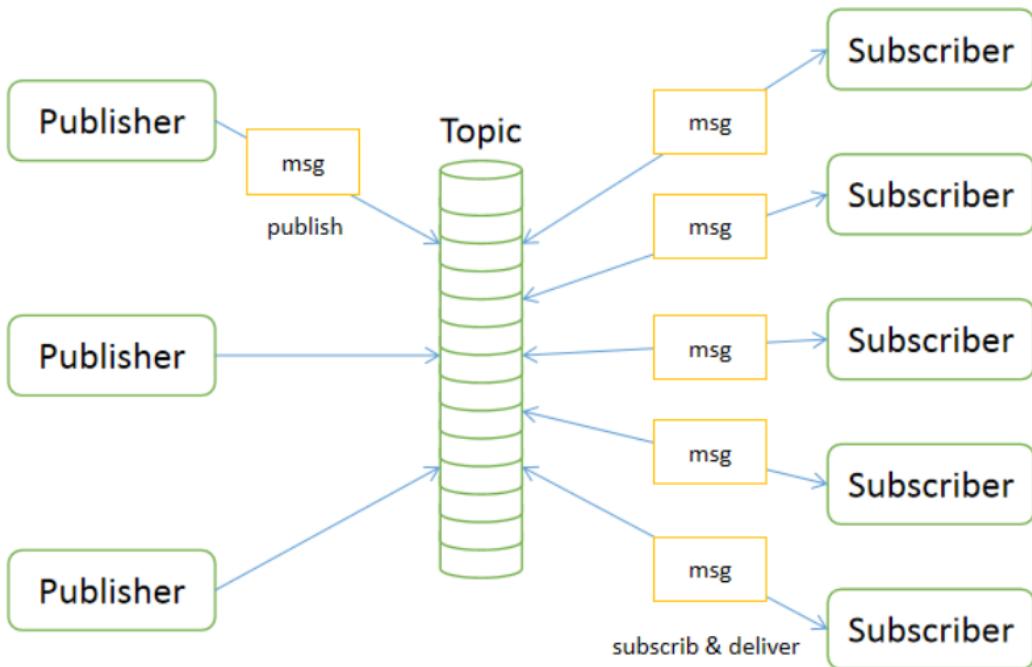
1.3.2 发布/订阅模式

发布/订阅模式下包括三个角色：

- 角色主题 (Topic)

- 发布者(Publisher)
- 订阅者(Subscriber)

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的消息会被多个订阅者消费。



发布/订阅模式特点：

- 每个消息可以有多个订阅者；
- 发布者和订阅者之间有时间上的依赖性
- 为了消费消息，订阅者必须保持在线运行。

1.4 消息队列实现机制

1.4.1 JMS

JMS (JAVA Message Service, Java消息服务) 是一个Java平台中关于面向消息中间件的API

允许应用程序组件基于JavaEE平台创建、发送、接收和读取消息

是一个消息服务的标准或者说是规范，是 Java 平台上有关面向消息中间件的技术规范

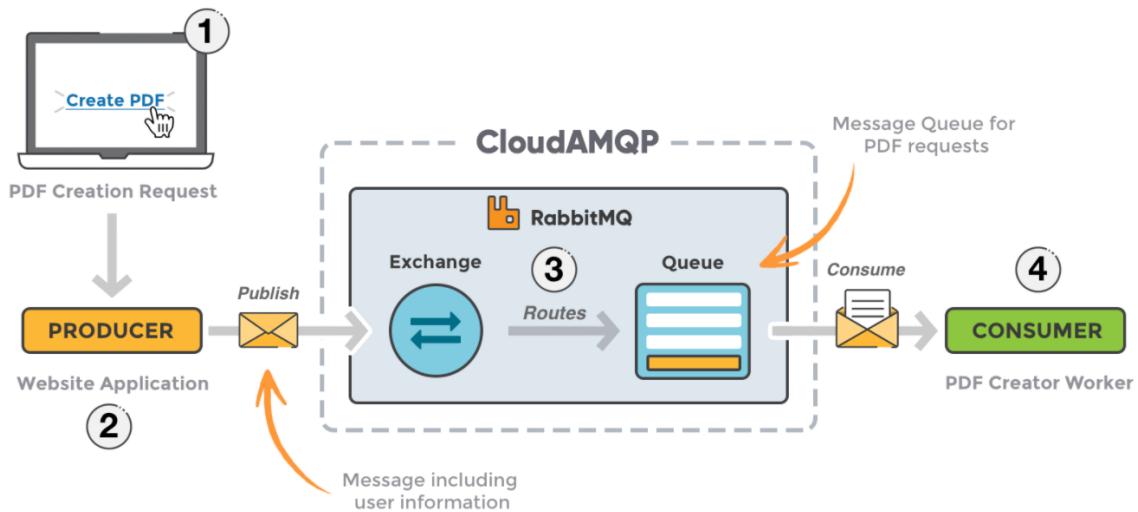
便于消息系统中的 Java 应用程序进行消息交换，并且通过提供标准的产生、发送、接收消息的接口，简化企业应用的开发。

JMS 消息机制主要分为两种模型：PTP 模型和 Pub/Sub 模型。

实现产品：Apache ActiveMQ

1.4.2 AMQP

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，不同的开发语言等条件的限制。Erlang中的实现有RabbitMQ等。



1.4.2 JMS VS AMQP

	JMS	AMQP
定义	Java api	Wire-protocol
跨语言	否	是
跨平台	否	是
Model	提供两种消息模型/模式：(1)、Peer-2-Peer (2)、Pub/sub	提供了五种消息模型：(1)、direct exchange (2)、fanout exchange (3)、topic change (4)、headers exchange (5)、system exchange本质来讲，后四种和JMS的pub/sub模型没有太大差别，仅是在路由机制上做了更详细的划分；
支持消息类型	多种消息类型：TextMessage、MapMessage、BytesMessage、StreamMessage、ObjectMessage、Message（只有消息头和属性）	byte[]当实际应用时，有复杂的消息，可以将消息序列化后发送。
综合评价	JMS 定义了JAVA API层面的标准；在java体系中，多个client均可以通过JMS进行交互，不需要应用修改代码，但是其对跨平台的支持较差；	AMQP定义了wire-level层的协议标准；天然具有跨平台、跨语言特性。

1.5常见的消息队列产品

RabbitMQ

[RabbitMQ](#) 2007年发布，是一个在[AMQP](#)(高级消息队列协议)基础上完成的，可复用的企业消息系统，是当前最主流的消息中间件之一。

ActiveMQ

[ActiveMQ](#)是由Apache出品，ActiveMQ 是一个完全支持JMS1.1和J2EE 1.4规范的 JMS Provider实现。它非常快速，支持多种语言的客户端和协议，而且可以非常容易的嵌入到企业的应用环境中，并有许多高级功能

RocketMQ

[RocketMQ](#)出自阿里公司的开源产品，用 Java 语言实现，在设计时参考了 Kafka，并做出了自己的一些改进，消息可靠性上比 Kafka 更好。RocketMQ在阿里集团被广泛应用在订单，交易，充值，流计算，消息推送，日志流式处理等

Kafka

[Apache Kafka](#)是一个分布式消息发布订阅系统。它最初由LinkedIn公司基于独特的设计实现为一个分布式的提交日志系统(a distributed commit log)，之后成为Apache项目的一部分。Kafka系统快速、可扩展并且可持久化。它的分区特性，可复制和可容错都是其不错的特性。

特性	ActiveMQ	RabbitMQ	Rocket MQ	kafka
开发语言	java	erlang	java	scala
单机吞吐量	万级	万级	10万级	10万级
时效性	ms级	us级	ms级	ms级以内
可用性	高(主从架构)	高(主从架构)	非常高 (分布式 架构)	非常高(分布式架构)
功能特性	成熟的产品，在很多公司得到应用；有较多的文档；各种协议支持较好	基于erlang开发，所以并发能力很强，性能极其好，延时很低；管理界面较丰富	MQ功能比较完备，扩展性佳	只支持主要的MQ功能，像一些消息查询，消息回溯等功能没有提供，毕竟是为大数据准备的，在大数据领域应用广。

综合上面的材料得出以下两点：

(1) 中小型软件公司：

建议选RabbitMQ。一方面，erlang语言天生具备高并发的特性，而且他的管理界面用起来十分方便。正所谓，成也萧何，败也萧何！他的弊端也在这里，虽然RabbitMQ是开源的，然而国内有几个能定制化开发erlang的程序员呢？所幸，RabbitMQ的社区十分活跃，可以解决开发过程中遇到的bug，这点对于中小型公司来说十分重要。不考虑rocketmq和kafka的原因是，一方面中小型软件公司不如互联网公司，数据量没那么大，选消息中间件，应首选功能比较完备的，所以kafka排除。不考虑rocketmq的原因是，rocketmq是阿里出品，如果阿里放弃维护rocketmq，中小型公司一般抽不出人来进行rocketmq的定制化开发，因此不推荐。

(2) 大型软件公司：

根据具体使用在rocketMq和kafka之间二选一。一方面，大型软件公司，具备足够的资金搭建分布式环境，也具备足够大的数据量。针对rocketMQ，大型软件公司也可以抽出人手对rocketMQ进行定制化开发，毕竟国内有能力改JAVA源码的人，还是相当多的。至于kafka，根据业务场景选择，如果有日志采集功能，肯定是首选kafka了。

2. kafka的基本介绍

2.1 什么是Kafka

官网：<http://kafka.apache.org/>



APACHE KAFKA

More than 80% of all Fortune 100 companies trust, and use Kafka.

Apache Kafka is an open-source distributed event streaming platform used by thousands of companies for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications.

Kafka是最初由Linkedin公司开发，是一个分布式、分区的、多副本的、多订阅者，基于zookeeper协调的分布式日志系统（也可以当做MQ系统），常见可以用于web/nginx日志、访问日志、消息服务等等，Linkedin于2010年贡献给了Apache基金会并成为顶级开源项目。

主要应用场景是：日志收集系统和消息系统。

Kafka主要设计目标如下：

- **以时间复杂度为O(1)的方式**提供消息持久化能力，即使对TB级以上数据也能保证常数时间的访问性能。
 - 算法复杂度：时间复杂度和空间复杂度

- 以时间复杂度为O(1)的方式：常数时间运行和数据量的增长无关，假如操作一个链表，那么无论链表的大还是小，操作时间是一样的
- 高吞吐率。即使在非常廉价的商用机器上也能做到单机支持每秒100K条消息的传输。
 - 支持普通服务器每秒百万级写入请求
 - Memory mapped Files
- 支持Kafka Server间的消息分区，及分布式消费，同时保证每个partition内的消息顺序传输。
- 同时支持离线数据处理和实时数据处理。
- Scale out:支持在线水平扩展

2.2 kafka的特点

- (1) 解耦。Kafka具备消息系统的优点，只要生产者和消费者数据两端遵循接口约束，就可以自行扩展或修改数据处理的业务过程。
- (2) 高吞吐量、低延迟。即使在非常廉价的机器上，Kafka也能做到每秒处理几十万条消息，而它的延迟最低只有几毫秒。
- (3) 持久性。Kafka可以将消息直接持久化在普通磁盘上，且磁盘读写性能优异。
- (4) 扩展性。Kafka集群支持热扩展，Kafka集群启动运行后，用户可以直接向集群添。
- (5) 容错性。Kafka会将数据备份到多台服务器节点中，即使Kafka集群中的某一台加新的Kafka服务节点宕机，也不会影响整个系统的功能。
- (6) 支持多种客户端语言。Kafka支持Java、.NET、PHP、Python等多种语言。
- (7) 支持多生产者和多消费者。

2.3 kafka的主要应用场景

- 消息处理 (MQ)

Kafka可以代替传统的消息队列软件，使用Kafka来实现队列有如下优点

- Kafka的append来实现消息的追加，保证消息都是有序的有先来后到的顺序，
- 稳定性强队列在使用中最怕丢失数据，Kafka能做到理论上的写成功不丢失
- 分布式容灾好
- 容量大相对于内存队列，Kafka的容量受硬盘影响
- 数据量不会影响到Kafka的速度

- 分布式日志系统(Log)

在很多时候我们需要对一些庞大的数据进行存留，日志存储这块会遇到巨大的问题，日志不能丢，日志存文件不好找，定位一条消息成本高（遍历当天日志文件），实时显示给用户难，这几类问题Kafka都能游刃有余

- Kafka的集群备份机制能做到n/2的可用，当n/2以下的机器宕机时存储的日志不会丢失
- Kafka可以对消息进行分组分片
- Kafka非常容易做到实时日志查询

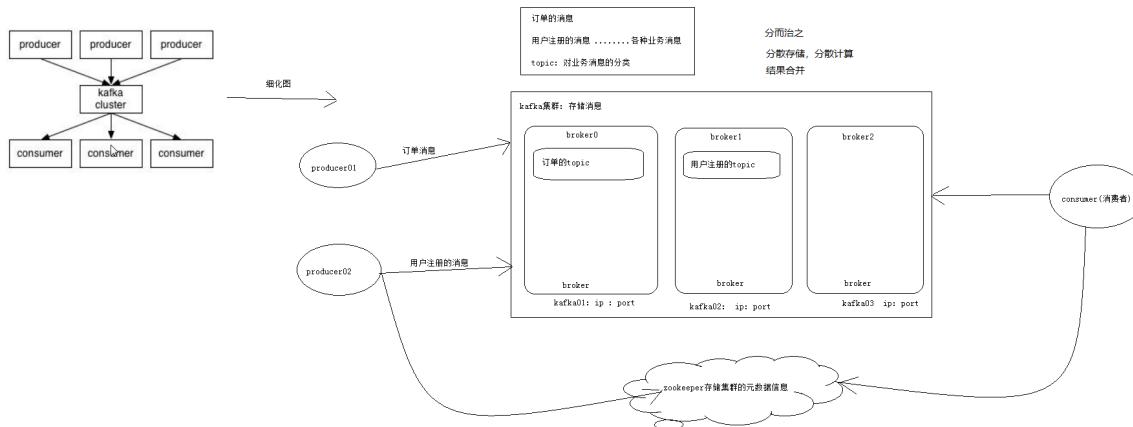
- 流式处理

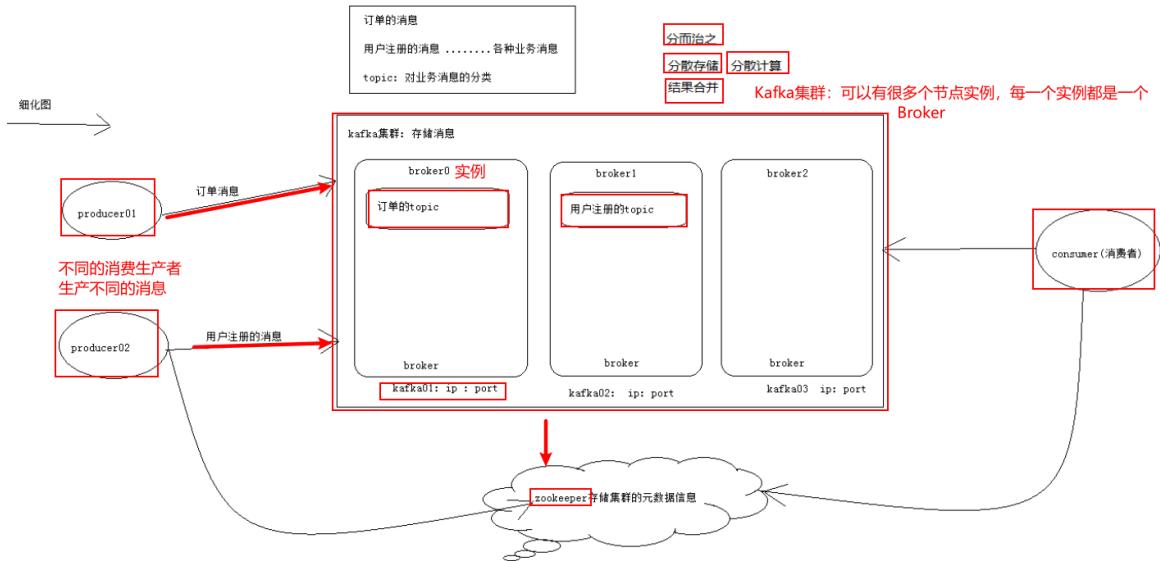
流式处理就是指实时地处理一个或多个事件流。

流式的处理框架(spark, storm, flink)从主题中读取数据，对其进行处理，并将处理后的结果数据写入新的主题，供用户和应用程序使用，kafka的强耐久性在流处理的上下文中也非常的有用

3. kafka的架构

3.1 架构案例





Kafka Cluster: 由多个服务器组成。每个服务器单独的名字**broker** (掮客)。

kafka broker: kafka集群中包含的服务器

Kafka Producer: 消息生产者、发布消息到 kafka 集群的终端或服务。

Kafka consumer: 消息消费者、负责消费数据。

Kafka Topic: 主题，一类消息的名称。存储数据时将一类数据存放在某个topic下，消费数据也是消费一类数据。

订单系统：创建一个topic，叫做order。

用户系统：创建一个topic，叫做user。

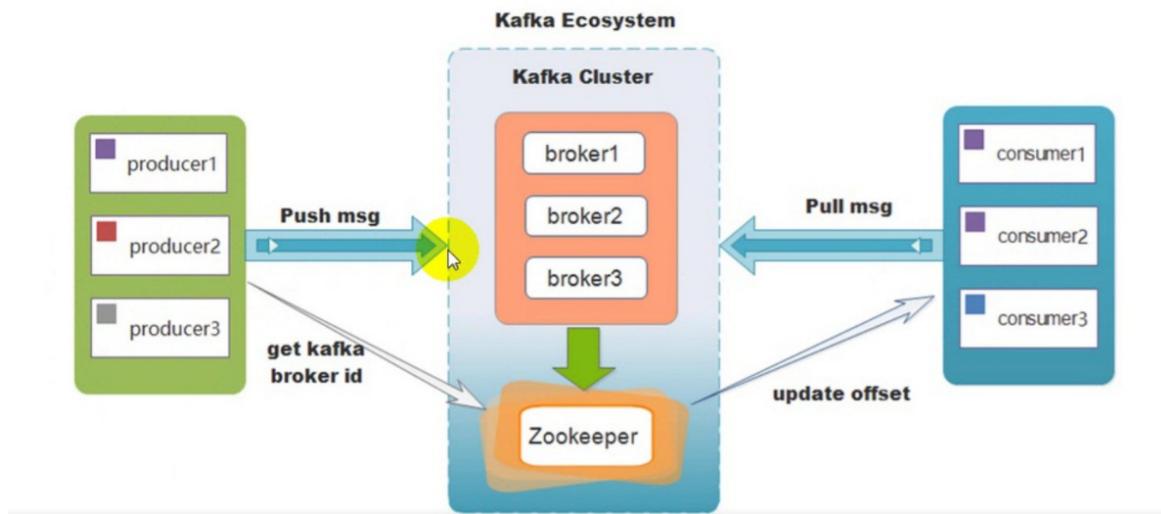
商品系统：创建一个topic，叫做product。

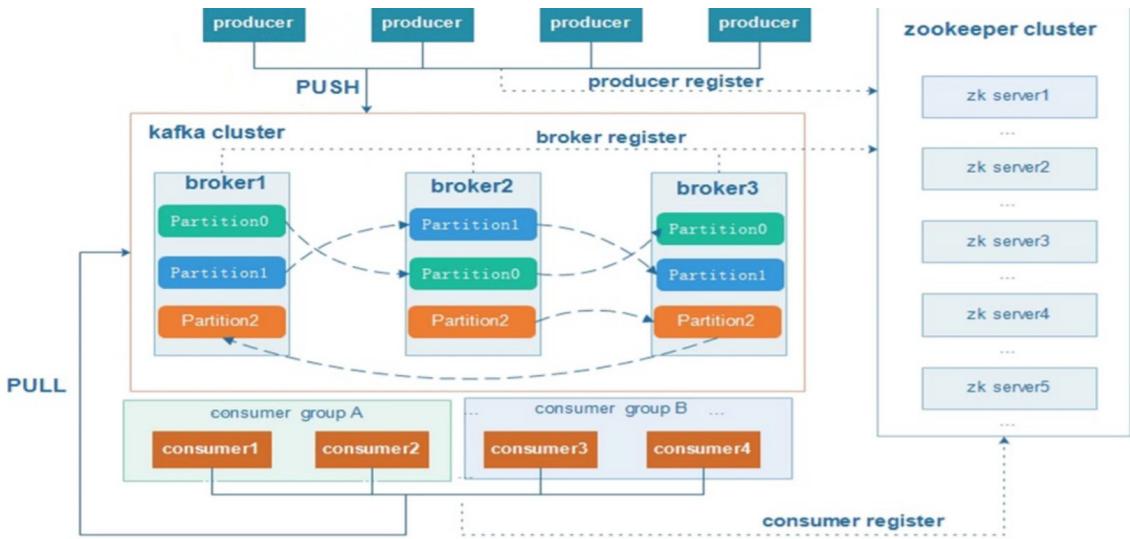
注意：Kafka的元数据都是存放在zookeeper中。

3.2 架构剖析

kafka架构的内部细节剖析：

Kafka架构





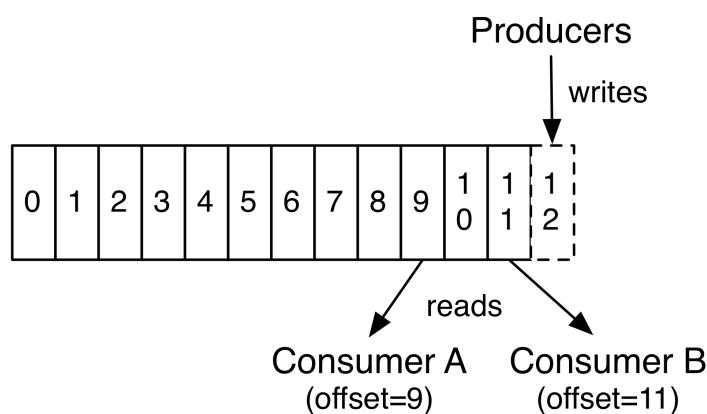
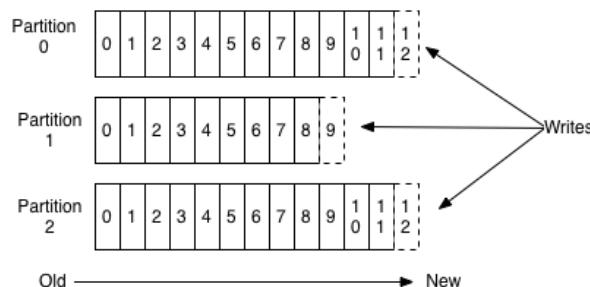
说明：kafka支持消息持久化，消费端为拉模型来拉取数据，消费状态和订阅关系有客户端负责维护，消息消费完后，不会立即删除，会保留历史消息。因此支持多订阅时，消息只会存储一份就可以了。

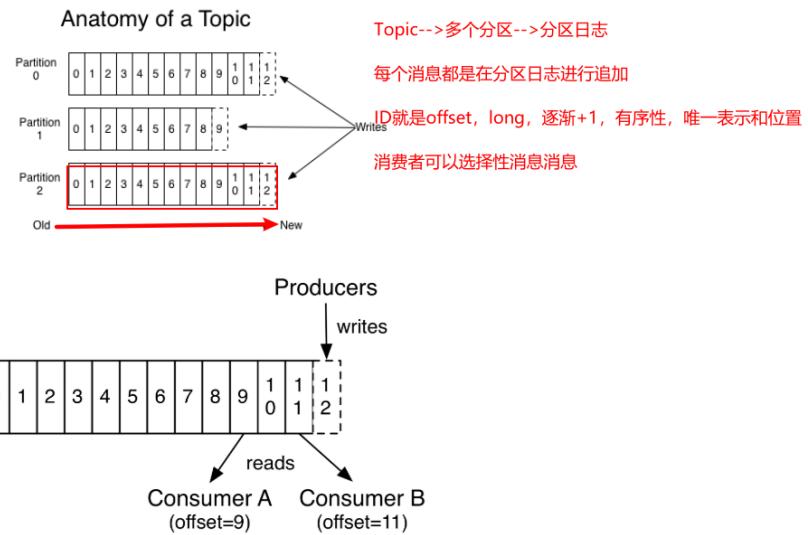
- Broker: kafka集群中包含一个或者多个服务实例，这种服务实例被称为Broker
- Topic: 每条发布到kafka集群的消息都有一个类别，这个类别就叫做Topic
- Partition: 分区，物理上的概念，每个topic包含一个或多个partition，一个partition对应一个文件夹，这个文件夹下存储partition的数据和索引文件，每个partition内部是有序的

3.3 关系解释

- Topic & Partition
 - 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。
 - 一个topic为一类消息，每条消息必须指定一个topic。
 - 对于每一个topic，Kafka集群都会维持一个分区日志。如下图
 - 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，称之为offset，offset用来唯一的标识分区中每一条记录。

Anatomy of a Topic





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

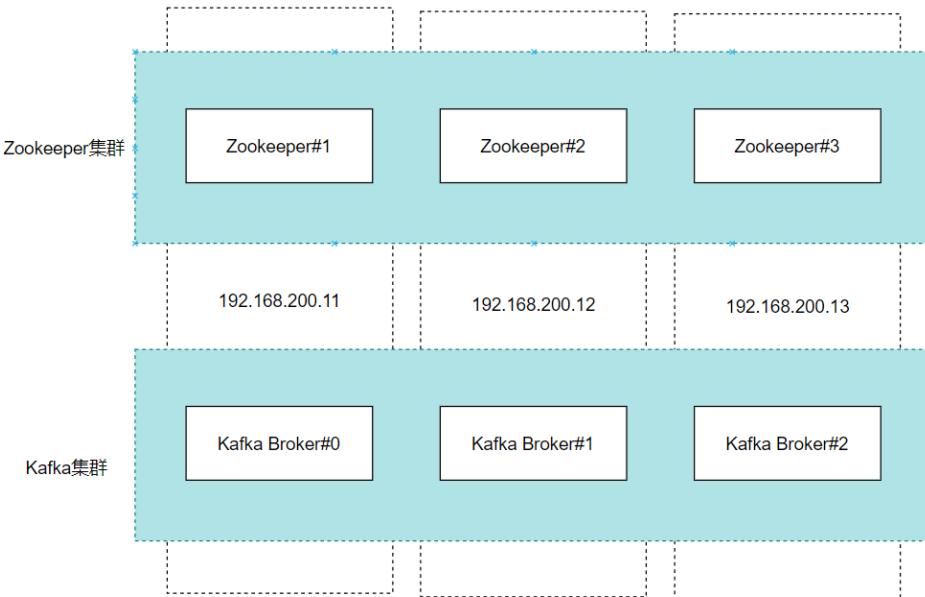
，但是实际上，由于这个位置由消费者控制，所以消费者可以采用任何顺序来消费记录。例如，一个消费者可以重置到一个旧的偏移量，从而重新处理过去的数

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。

4. kafka集群环境搭建

ZooKeeper 作为给分布式系统提供协调服务的工具被 kafka 所依赖。在分布式系统中，消费者需要知道有哪些生产者是可用的，而如果每次消费者都需要和生产者建立连接并测试是否成功连接，那效率也太低了，显然是不可取的。而通过使用 ZooKeeper 协调服务，Kafka 就能将 Producer, Consumer, Broker 等结合在一起，同时借助 ZooKeeper，Kafka 就能够将所有组件在无状态的条件下建立起生产者和消费者的订阅关系，实现负载均衡。



4.1 准备工作

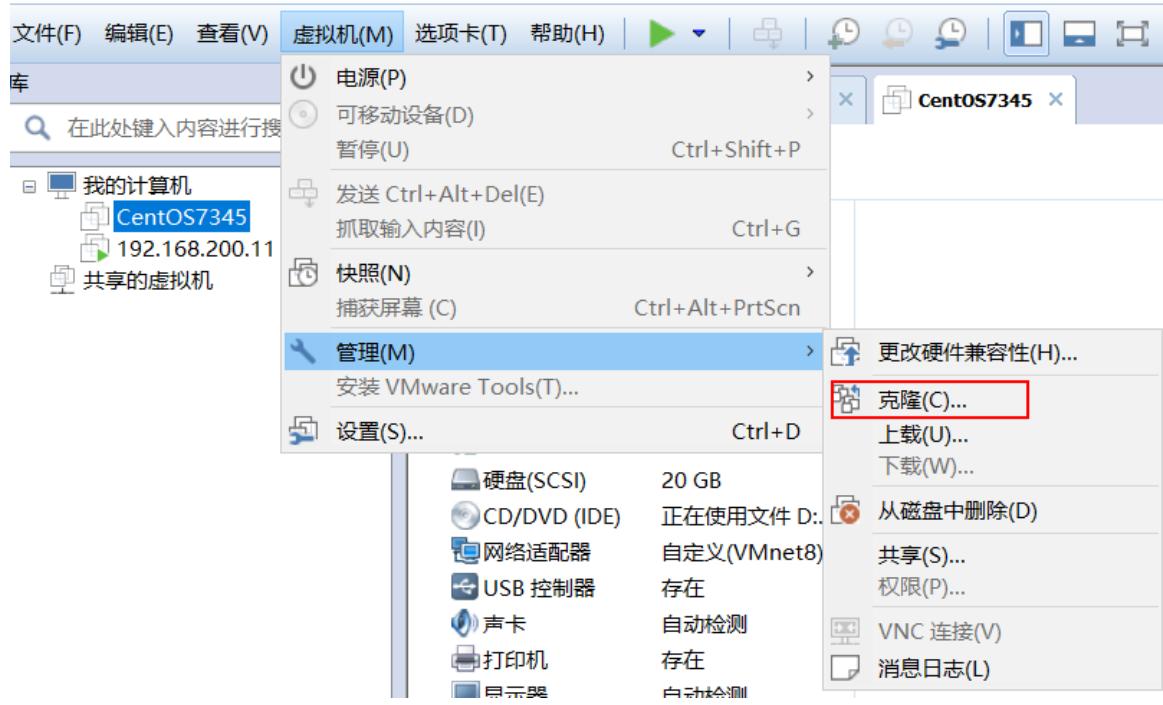
环境准备：

准备三台服务器, 安装jdk1.8 ,其中每一台虚拟机的hosts文件中都需要配置如下的内容

```
192.168.200.11 node1
192.168.200.12 node2
192.168.200.13 node3
```

实现方式：

1. 将原有学习Docker时的Centos7克隆，克隆出一台虚拟机



克隆源

您想从哪个状态创建克隆？

克隆自

虚拟机中的当前状态 (C)

从当前状态创建链接克隆将创建一个新快照。

现有快照(仅限关闭的虚拟机)(S):

此虚拟机没有可克隆的快照。

< 上一步 (B)

下一步 (N) >

取消

克隆类型

您希望如何克隆此虚拟机？

克隆方法

创建链接克隆(L)

链接克隆是对原始虚拟机的引用，所需的存储磁盘空间较少。但是，必须能够访问原始虚拟机才能运行。

创建完整克隆(F)

完整克隆是原始虚拟机当前状态的完整副本。此副本虚拟机完全独立，但需要较多的存储磁盘空间。

< 上一步(B)

下一步(N) >

取消

克隆虚拟机向导

X

新虚拟机名称

您要为此虚拟机使用什么名称？

虚拟机名称(V)

192.168.200.11 - node1

指定虚拟机名称

位置(L)

C:\Users\jetwu\Documents\Virtual Machines\192.168.200.11 -

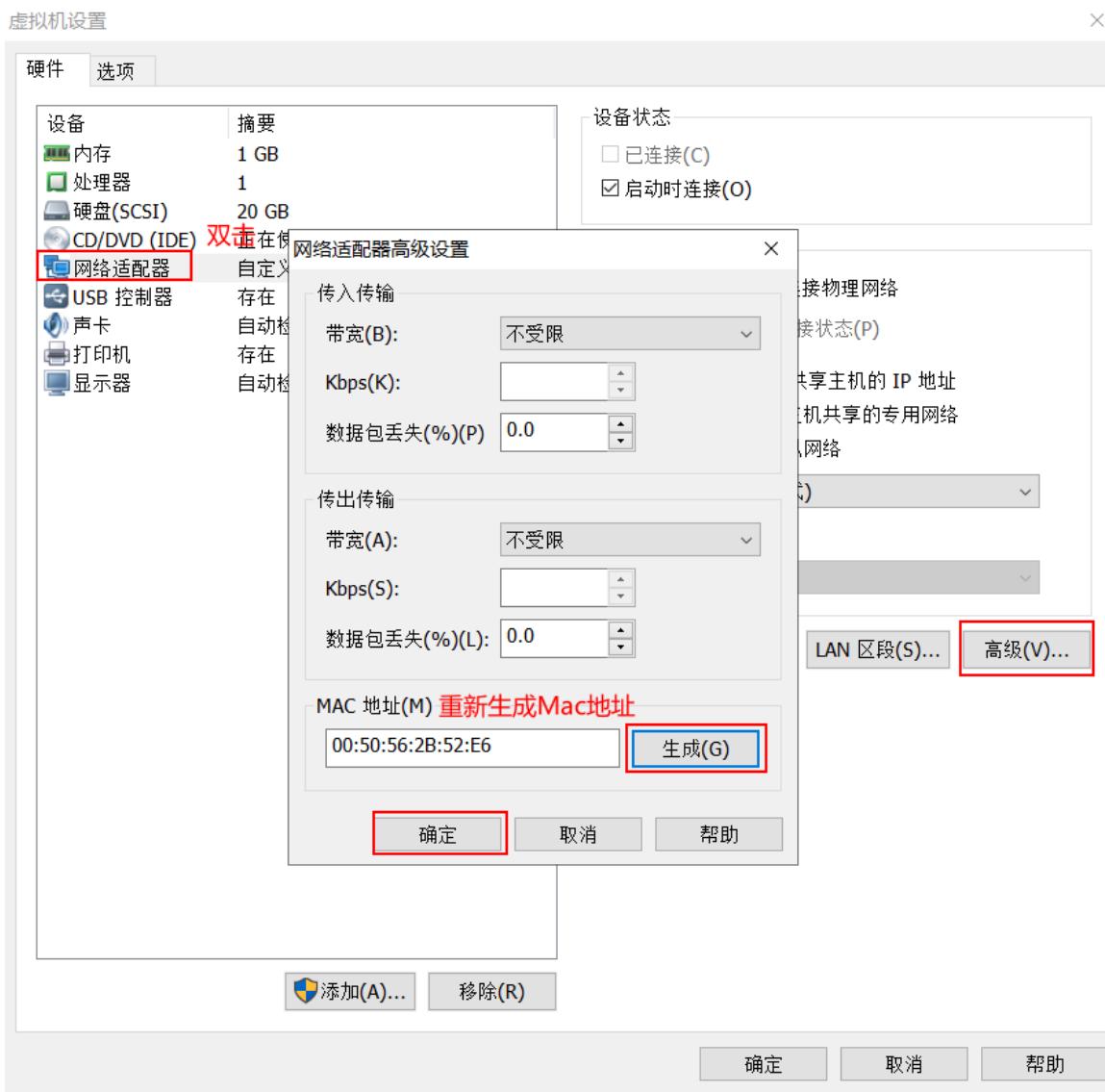
[浏览\(R\)...](#)

指定虚拟机文件存放位置，不要冲突

< 上一步(B)

完成

取消



2、修改IP地址

修改网卡配置文件 vi /etc/sysconfig/network-scripts/ifcfg-ens33

- (1) bootproto=static, 表示使用静态IP
- (2) onboot=yes, 表示将网卡设置为开机启用
- (3) 将原有的原有IP修改为192.168.200.11

```

TYPE=Ethernet
PROXY_METHOD=none
BROWSER_ONLY=no
BOOTPROTO=static
DEFROUTE=yes
IPV4_FAILURE_FATAL=no
IPV6INIT=yes
IPV6_AUTOCONF=yes
IPV6_DEFROUTE=yes
IPV6_FAILURE_FATAL=no
IPV6_ADDR_GEN_MODE=stable-privacy
NAME=ens33
UUID=b8fd5718-51f5-48f8-979b-b9f1f7a5ebf2
DEVICE=ens33
ONBOOT=yes
IPADDR=192.168.200.11
GATEWAY=192.168.200.2
NETMASK=255.255.255.0
NM_CONTROLLED=no
DNS1=8.8.8.8
DNS2=8.8.4.4

```

- (4) 重启网络服务

```
service network restart
```

(5) 安装目录创建

```
安装包存放的目录: /export/software  
安装程序存放的目录: /export/servers  
数据目录: /export/data  
日志目录: /export/logs
```

```
创建各级目录命令:  
mkdir -p /export/servers/  
mkdir -p /export/software/  
mkdir -p /export/data/  
mkdir -p /export/logs/
```

(6) 修改host

执行命令“`cd /etc/`”进入服务器etc目录；

执行命令“`vi hosts`”编辑hosts文件；

输入你要修改的内容：

```
127.0.0.1 localhost localhost.localdomain localhost4 localhost4.localdomain4  
::1 localhost localhost.localdomain localhost6 localhost6.localdomain6  
  
192.168.200.11 node1  
192.168.200.12 node2  
192.168.200.13 node3
```

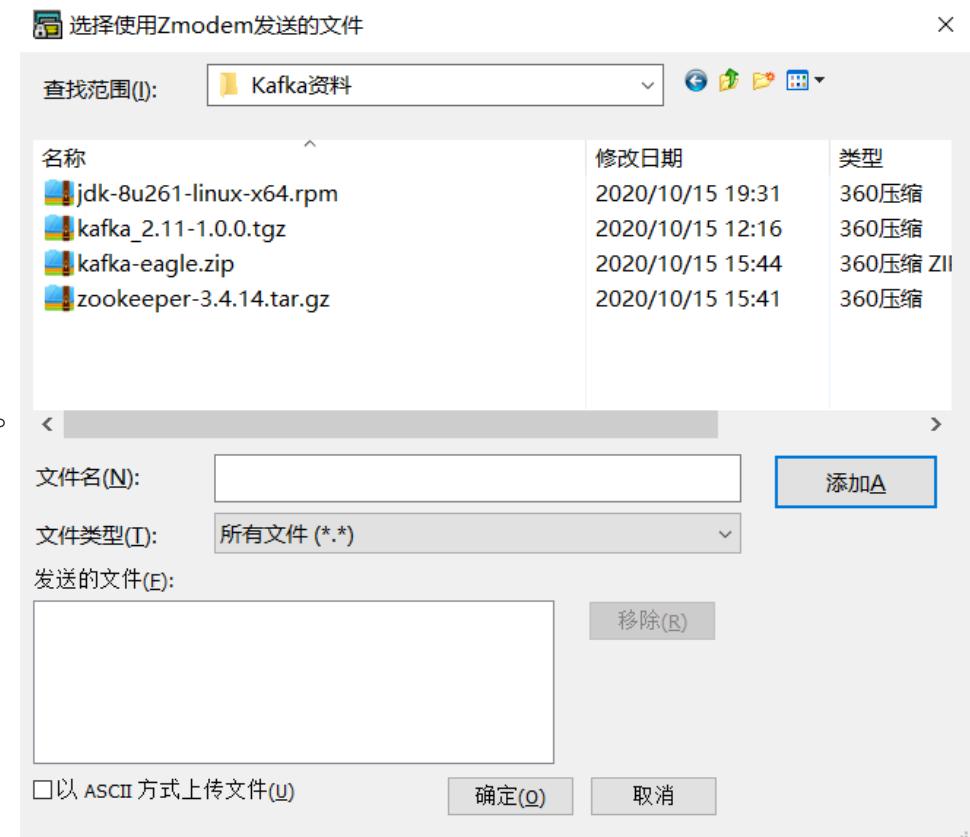
执行命令“`/etc/init.d/network restart`”重启hosts；

执行命令“`cat /etc/hosts`”可以查看到hosts文件修改成功。

4.2 Zookeeper集群搭建

1. Linux安装JDK，三台Linux都安装。

- 上传JDK到linux
 - 上传文件两种方式：使用SSH方式、使用CRT方式
- 使用CRT需要先在Linux虚拟机上安装lrsz上传工具，安装方式：`yum install -y lrssz`
 - 安装lrssz之后，是需要在Linux命令行中输入：rz，就可以弹出一个文件上传窗口



- 安装并配置JDK

```

# 使用rpm安装JDK
rpm -ivh jdk-8u261-linux-x64.rpm
# 默认的安装路径是/usr/java/jdk1.8.0_261-amd64
# 配置JAVA_HOME
vi /etc/profile
# 文件最后添加两行
export JAVA_HOME=/usr/java/jdk1.8.0_261-amd64
export PATH=$PATH:$JAVA_HOME/bin
# 退出vi, 使配置生效
source /etc/profile

```

- 查看JDK是否正确安装

```
java -version
```

```

[root@node2 ~]# java -version
java version "1.8.0_261"
Java(TM) SE Runtime Environment (build 1.8.0_261-b12)
Java HotSpot(TM) 64-Bit Server VM (build 25.261-b12, mixed mode)
[root@node2 ~]#

```

2. Linux 安装Zookeeper, 三台Linux都安装, 以搭建Zookeeper集群

- 上传zookeeper-3.4.14.tar.gz
- 解压并配置zookeeper (配置data目录, 集群节点)

```

# node1操作
# 解压到/opt目录
tar -zxf zookeeper-3.4.14.tar.gz -C /opt
# 配置
cd /opt/zookeeper-3.4.14/conf
# 配置文件重命名后生效
cp zoo_sample.cfg zoo.cfg
#编辑
vi zoo.cfg
# 设置数据目录
dataDir=/var/lagou/zookeeper/data
# 添加
server.1=node1:2881:3881
server.2=node2:2881:3881
server.3=node3:2881:3881

# 退出vim
mkdir -p /var/lagou/zookeeper/data
echo 1 > /var/lagou/zookeeper/data/myid

# 配置环境变量
vi /etc/profile
# 添加
export ZOOKEEPER_PREFIX=/opt/zookeeper-3.4.14
export PATH=$PATH:$ZOOKEEPER_PREFIX/bin
export ZOO_LOG_DIR=/var/lagou/zookeeper/log

# 退出vim, 让配置生效
source /etc/profile

```

- node2配置

```

# 配置环境变量
vim /etc/profile
# 在配置JDK环境变量基础上, 添加内容
export ZOOKEEPER_PREFIX=/opt/zookeeper-3.4.14
export PATH=$PATH:$ZOOKEEPER_PREFIX/bin
export ZOO_LOG_DIR=/var/lagou/zookeeper/log

# 退出vim, 让配置生效
source /etc/profile

mkdir -p /var/lagou/zookeeper/data
echo 2 > /var/lagou/zookeeper/data/myid

```

- node3配置

```

# 配置环境变量
vim /etc/profile
# 在配置JDK环境变量基础上, 添加内容
export ZOOKEEPER_PREFIX=/opt/zookeeper-3.4.14
export PATH=$PATH:$ZOOKEEPER_PREFIX/bin
export ZOO_LOG_DIR=/var/lagou/zookeeper/log

# 退出vim, 让配置生效
source /etc/profile

mkdir -p /var/lagou/zookeeper/data
echo 3 > /var/lagou/zookeeper/data/myid

```

- 启动zookeeper

```

# 在三台Linux上启动zookeeper
[root@node1 ~]# zkServer.sh start
[root@node2 ~]# zkServer.sh start
[root@node3 ~]# zkServer.sh start

# 在三台Linux上查看Zookeeper的状态
[root@node1 ~]# zkServer.sh status
ZooKeeper JMX enabled by default
Using config: /opt/zookeeper-3.4.14/bin/..../conf/zoo.cfg
Mode: follower

[root@node2 ~]# zkServer.sh status
ZooKeeper JMX enabled by default
Using config: /opt/zookeeper-3.4.14/bin/..../conf/zoo.cfg
Mode: leader

[root@node3 ~]# zkServer.sh status
ZooKeeper JMX enabled by default
Using config: /opt/zookeeper-3.4.14/bin/..../conf/zoo.cfg
Mode: follower

```

4.3 下载安装包

中文网站: <http://kafka.apache.org/>

英文网站: <http://kafka.apache.org/>

1.0.0

- Released November 1, 2011
- Source download: [kafka-1.0.0-src.tgz \(asc, sha512\)](#)
- Binary downloads:
 - Scala 2.11 - [kafka_2.11-1.0.0.tgz \(asc, sha512\)](#) 推荐使用这个
 - Scala 2.12 - [kafka_2.12-1.0.0.tgz \(asc, sha512\)](#)

We build for multiple versions of Scala. This only matters if you are using Scala and you want a version built for the same Scala version you use. Otherwise any version should work (2.11 is recommended).

由于kafka是scala语言编写的，基于scala的多个版本，kafka发布了多个版本。

其中2.11是推荐版本.

- 推荐直接使用资料中的版本即可:

kafka资料 > 资料			
名称	修改日期	类型	
kafka_2.11-1.0.0.tgz	2020/10/15 12:16	WinR	
kafka-eagle.zip	2020/10/15 15:44	WinR	
zookeeper-3.4.14.tar.gz	2020/10/15 15:41	WinR	

4.4 上传安装包并解压

```

使用 rz 命令将安装包上传至 /export/software
1) 切换目录上传安装包
cd /export/software
rz # 选择对应安装包上传即可

2) 解压安装包到指定目录下
tar -zvxf kafka_2.11-1.0.0.tgz -C /export/servers/
cd /export/servers/

3) 重命名(由于名称太长)
mv kafka_2.11-1.0.0 kafka

```

4.5 修改kafka的核心配置文件

```

cd /export/servers/kafka/config/
vi server.properties

主要修改一下6个地方:
1) broker.id 需要保证每一台kafka都有一个独立的broker
2) log.dirs 数据存放的目录
3) zookeeper.connect zookeeper的连接地址信息
4) delete.topic.enable 是否直接删除topic
5) host.name 主机的名称
6) 修改: listeners=PLAINTEXT://node1:9092
#broker.id 标识了kafka集群中一个唯一broker。
broker.id=0
num.network.threads=3
num.io.threads=8
socket.send.buffer.bytes=102400
socket.receive.buffer.bytes=102400
socket.request.max.bytes=104857600

# 存放生产者生产的数据 数据一般以topic的方式存放
log.dirs=/export/data/kafka
num.partitions=1
num.recovery.threads.per.data.dir=1
offsets.topic.replication.factor=1
transaction.state.log.replication.factor=1
transaction.state.log.min.isr=1
log.retention.hours=168
log.segment.bytes=1073741824
log.retention.check.interval.ms=300000

# zk的信息
zookeeper.connect=node1:2181,node2:2181,node3:2181
zookeeper.connection.timeout.ms=6000
group.initial.rebalance.delay.ms=0

delete.topic.enable=true
host.name=node1

```

4.6 将配置好的kafka分发到其他二台主机

```

cd /export/servers
scp -r kafka/ node2:$PWD
scp -r kafka/ node3:$PWD

```

Linux scp 命令用于 Linux 之间复制文件和目录。

scp 是 secure copy 的缩写, scp 是 linux 系统下基于 ssh 登陆进行安全的远程文件拷贝命令。

- 拷贝后, 需要修改每一台的broker.id 和 host.name和listeners

```

ip为11的服务器: broker.id=0 , host.name=node1  listeners=PLAINTEXT://node1:9092
ip为12的服务器: broker.id=1 , host.name=node2  listeners=PLAINTEXT://node2:9092
ip为13的服务器: broker.id=2 , host.name=node3  listeners=PLAINTEXT://node3:9092

```

- 在每一台的服务器执行创建数据文件的命令

```

mkdir -p /export/data/kafka

```

4.7 启动集群

注意事项：在kafka启动前，一定要让zookeeper启动起来

```
cd /export/servers/kafka/bin  
#前台启动  
. ./kafka-server-start.sh /export/servers/kafka/config/server.properties  
  
#后台启动  
nohup ./kafka-server-start.sh /export/servers/kafka/config/server.properties 2>&1 &
```

注意：可以启动一台**broker**，单机版。也可以同时启动三台**broker**，组成一个**kafka**集群版

```
#kafka停止  
. ./kafka-server-stop.sh
```

可以通过 jps 查看 kafka 进程是否已经启动了

登录的前提是，通过 **jps** 是可以看到 **kafka** 的进程
登录 **zookeeper**: zkCli.sh
执行: ls /brokers/ids

```
WatchedEvent state:SyncConnected type:None path:null  
[zk: localhost:2181(CONNECTED) 0] ls /  
[cluster, controller, test02, brokers, zookeeper, test01, admin, isr_change_notification, log_dir_event_notification, controller_epoch, consumers, latest_producer_id_block, config, hbase]  
[zk: localhost:2181(CONNECTED) 1] ls /brokers  
[ids, topics, seqid]  
[zk: localhost:2181(CONNECTED) 2] ls /brokers/ids  
[0, 1, 2]  
[zk: localhost:2181(CONNECTED) 3]
```

4.8 Docker环境下的Kafka集群搭建

hostname	ip addr	port	listener
zoo1	192.168.0.11	2184:2181	
zoo2	192.168.0.12	2185:2181	
zoo3	192.168.0.13	2186:2181	
kafka1	192.168.0.14	9092:9092	kafka1
kafka2	192.168.0.15	9093:9092	kafka1
kafka3	192.168.0.16	9094:9092	kafka1
kafka-manager	192.168.0.17	9000:9000	
宿主机	192.168.200.20		

4.8.1 准备工作

1) 克隆VM，修改IP地址为192.168.200.20

修改网络配置: vi /etc/sysconfig/network-scripts/ifcfg-ens33

```
TYPE=Ethernet  
PROXY_METHOD=none  
BROWSER_ONLY=no  
BOOTPROTO=static  
DEFROUTE=yes  
IPV4_FAILURE_FATAL=no  
IPV6INIT=yes  
IPV6_AUTOCONF=yes  
IPV6_DEFROUTE=yes  
IPV6_FAILURE_FATAL=no  
IPV6_ADDR_GEN_MODE=stable-privacy  
NAME=ens33  
UUID=b8fd5718-51f5-48f8-979b-b9f1f7a5ebf2  
DEVICE=ens33
```

```
ONBOOT=yes
IPADDR=192.168.200.20
GATEWAY=192.168.200.2
NETMASK=255.255.255.0
NM_CONTROLLED=no
DNS1=8.8.8.8
DNS2=8.8.4.4
```

2) 安装docker - compose

Compose 是用于定义和运行多容器 Docker 应用程序的工具。

如果我们还是使用原来的方式操作docker，那么就需要下载三个镜像：Zookeeper、Kafka、Kafka-Manager，需要对Zookeeper安装三次并配置集群、需要对Kafka安装三次，修改配置文件，Kafka-Manager安装一次，但是需要配置端口映射机器Zookeeper、Kafka容器的信息。但是引入Compose之后可以使用yaml格式的配置文件配置好这些信息，每个image只需要编写一个yaml文件，可以在文件中定义集群信息、端口映射等信息，运行该文件即可创建完成集群。

通过 Compose，您可以使用 YML 文件来配置应用程序需要的所有服务。然后，使用一个命令，就可以从 YML 文件配置中创建并启动所有服务。

Compose 使用的两个步骤：

- 使用 docker-compose.yml 定义构成应用程序的服务，这样它们可以在隔离环境中一起运行。
- 执行 docker-compose up 命令来启动并运行整个应用程序。

```
#curl 是一种命令行工具，作用是发出网络请求，然后获取数据
curl -L https://github.com/docker/compose/releases/download/1.8.0/run.sh > /usr/local/bin/docker-compose

#chmod (change mode) 命令是控制用户对文件的权限的命令
chmod +x /usr/local/bin/docker-compose

#查看版本
docker-compose --version
```

3) 拉取镜像

```
#拉取zookeeper镜像
docker pull zookeeper:3.4
#拉取kafka镜像
docker pull wurstmeister/kafka
#拉取kafka-manager镜像
docker pull sheepkiller/kafka-manager:latest
```

```
[root@localhost ~]# docker images
REPOSITORY          TAG      IMAGE ID      CREATED        SIZE
zookeeper           3.4      721354d41dae  5 days ago   257MB
wurstmeister/kafka  latest   40094a582680  2 months ago  435MB
sheepkiller/kafka-manager  latest   4e4a8c5dabab  2 years ago   463MB
docker/compose      1.8.0    89188432ef03  4 years ago   59.1MB
```

4) 创建集群网络

基于Linux宿主机而工作的，也是在Linux宿主机创建，创建之后Docker容器中的各个应用程序可以使用该网络。

```
#创建
docker network create --driver bridge --subnet 192.168.0.0/24 --gateway 192.168.0.1 kafka
#查看
docker network ls
```

5) 网络设置

新建网段之后可能会出现：WARNING: IPv4 forwarding is disabled. Networking will not work.

解决方式：

第一步：在宿主机上执行：echo "net.ipv4.ip_forward=1" >>/usr/lib/sysctl.d/00-system.conf

第二步：重启network和docker服务

```
[root@localhost ~]# systemctl restart network && systemctl restart docker
```

4.8.2 搭建过程

每个镜像一个Yml文件，Zookeeper、Kafka、Kafka-Manager一个

编写yml文件

1) docker-compose-zookeeper.yml

Zookeeper各个节点的信息，端口映射，集群信息，网络配置

```
version: '2'      #指定 compose 文件的版本

services: #通过镜像安装容器的配置
zool:
  image: zookeeper:3.4  #使用的镜像
  restart: always        #当Docker重启时，该容器重启
  hostname: zool         #类似于在基于Linux虚拟机kafka集群中hosts文件的值
  container_name: zool
  ports:
    - 2181:2181          #端口映射
  environment: #集群环境
    ZOO_MY_ID: 1 #当前zookeeper实例的id
    ZOO_SERVERS: server.1=0.0.0.0:2888:3888 server.2=zoo2:2888:3888 server.3=zoo3:2888:3888 #集群节点
  networks: #使用的网络配置
    kafka:
      ipv4_address: 192.168.0.11

zoo2:
  image: zookeeper:3.4
  restart: always
  hostname: zool
  container_name: zool
  ports:
    - 2181:2181
  environment:
    ZOO_MY_ID: 2
    ZOO_SERVERS: server.1=zoo1:2888:3888 server.2=0.0.0.0:2888:3888 server.3=zoo3:2888:3888
  networks:
    kafka:
      ipv4_address: 192.168.0.12

zoo3:
  image: zookeeper:3.4
  restart: always
  hostname: zool
  container_name: zool
  ports:
    - 2181:2181
  environment:
    ZOO_MY_ID: 3
    ZOO_SERVERS: server.1=zoo1:2888:3888 server.2=zoo2:2888:3888 server.3=0.0.0.0:2888:3888
  networks:
    kafka:
      ipv4_address: 192.168.0.13

networks:
  kafka:
    external:
      name: kafka
```

2) docker-compose-kafka.yml

```
version: '2'

services:
  kafka1:
    image: wurstmeister/kafka  #image
    restart: always
    hostname: kafka1
    container_name: kafka1
    privileged: true
    ports:
      - 9092:9092
    environment: #集群环境配置
      KAFKA_ADVERTISED_HOST_NAME: kafka1
      KAFKA_LISTENERS: PLAINTEXT://kafka1:9092
```

```

KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka1:9092
KAFKA_ADVERTISED_PORT: 9092
KAFKA_ZOOKEEPER_CONNECT: zoo1:2181,zoo2:2181,zoo3:2181
external_links: # 配置Zookeeper集群的地址
- zoo1
- zoo2
- zoo3
networks:
  kafka:
    ipv4_address: 192.168.0.14

kafka2:
  image: wurstmeister/kafka
  restart: always
  hostname: kafka2
  container_name: kafka2
  privileged: true
  ports:
  - 9093:9093
  environment:
    KAFKA_ADVERTISED_HOST_NAME: kafka2
    KAFKA_LISTENERS: PLAINTEXT://kafka2:9093
    KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka2:9093
    KAFKA_ADVERTISED_PORT: 9093
    KAFKA_ZOOKEEPER_CONNECT: zoo1:2181,zoo2:2181,zoo3:2181
  external_links:
  - zoo1
  - zoo2
  - zoo3
  networks:
    kafka:
      ipv4_address: 192.168.0.15

kafka3:
  image: wurstmeister/kafka
  restart: always
  hostname: kafka3
  container_name: kafka3
  privileged: true
  ports:
  - 9094:9094
  environment:
    KAFKA_ADVERTISED_HOST_NAME: kafka3
    KAFKA_LISTENERS: PLAINTEXT://kafka3:9094
    KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka3:9094
    KAFKA_ADVERTISED_PORT: 9094
    KAFKA_ZOOKEEPER_CONNECT: zoo1:2181,zoo2:2181,zoo3:2181
  external_links:
  - zoo1
  - zoo2
  - zoo3
  networks:
    kafka:
      ipv4_address: 192.168.0.16

networks:
  kafka:
    external:
      name: kafka

```

3) docker-compose-manager.yml

```

version: '2'

services:
  kafka-manager:
    image: sheepkiller/kafka-manager:latest
    restart: always
    container_name: kafka-manager
    hostname: kafka-manager
    ports:

```

```

    - 9000:9000
environment: #可以管理zoo集群和kafka集群
ZK_HOSTS: zoo1:2181,zoo2:2181,zoo3:2181
KAFKA_BROKERS: kafka1:9092,kafka2:9092,kafka3:9092
APPLICATION_SECRET: letmein
KM_ARGS: -Djava.net.preferIPv4Stack=true
networks:
  kafka:
    ipv4_address: 192.168.0.17

networks:
  kafka:
    external:
      name: kafka

```

2) 将yaml文件上传到Docker宿主机中

- 安装: yum install -y lrzs
- 上传到指定目录

3) 开始部署

使用命令: docker-compose up -d

参数说明: up表示启动, -d表示后台运行

docker-compose -f /home/docker-compose-zookeeper.yml up -d

参数说明: -f:表示加载指定位置的yaml文件

docker-compose -f /home/docker-compose-kafka.yml up -d

docker-compose -f /home/docker-compose-manager.yml up -d

4) 测试

浏览器访问宿主机: <http://192.168.200.20:9000/>

测试:

5. kafka的基本操作

在docker环境中演示

(1) 创建topic

创建一个名字为test的主题, 有一个分区, 有三个副本。一个主题下可以有多个分区, 每个分区可以用对应的副本。

Docker:

```

#登录到Kafka容器
docker exec -it 9218e985e160 /bin/bash
#切换到bin目录
cd opt/kafka/bin/
#执行创建
kafka-topics.sh --create --zookeeper zoo1:2181 --replication-factor 3 --partitions 1 --topic test

```

--create: 新建命令

--zookeeper: Zookeeper节点, 一个或多个

--replication-factor: 指定副本, 每个分区有三个副本。

--partitions: 1

Partition	Latest Offset	Leader	Replicas	In Sync Replicas	Preferred Leader?	Under Replicated?
0		1001	(1001,1002,1003)	(1001,1002,1003)	true	false
1		1002	(1002,1003,1001)	(1002,1003,1001)	true	false
2		1003	(1003,1001,1002)	(1003,1001,1002)	true	false

(2) 查看主题命令

查看kafka当中存在的主题

```
kafka-topics.sh --list --zookeeper zoo1:2181,zoo2:2181,zoo3:2181
```

_consumer_offsets 这个topic是由kafka自动创建的，默认50个分区，存储消费位移信息（offset），老版本架构中是存储在Zookeeper 中。

The screenshot shows the Kafka UI interface for the __consumer_offsets topic. On the left, there's a 'Topic Summary' table with various metrics like Replication (1), Number of Partitions (50), and Total number of Brokers (3). Below it is a 'Config' table with entries for segment.bytes, compression.type, and cleanup.policy. On the right, there's an 'Operations' section with buttons for Delete Topic, Reassign Partitions, Generate Partition Assignments, Add Partitions, Update Config, and Manual Partition Assignments. The main area displays 'Partitions by Broker' with three brokers (1001, 1002, 1003) each having 17 partitions. The partitions are listed as ranges of offsets.

Topic Summary	
Replication	1
Number of Partitions	50
Sum of partition offsets	0
Total number of Brokers	3
Number of Brokers for Topic	3
Preferred Replicas %	100
Brokers Skewed %	0
Brokers Spread %	100
Under-replicated %	0
Config	Value
segment.bytes	104857600
compression.type	producer
cleanup.policy	compact

Broker	# of Partitions	Partitions	Skewed?
1001	17	(0,3,6,9,12,15,18,21,24,27,30,33,36,39,42,45,48)	false
1002	17	(1,4,7,10,13,16,19,22,25,28,31,34,37,40,43,46,49)	false
1003	16	(2,5,8,11,14,17,20,23,26,29,32,35,38,41,44,47)	false

(3) 生产者生产数据

模拟生产者来生产数据：

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。

```
kafka-console-producer.sh --broker-list kafka1:9092,kafka2:9093,kafka3:9094 --topic test
This is a message
This is another message
```

(4) 消费者消费数据

```
kafka-console-consumer.sh --bootstrap-server kafka1:9092, kafka2:9093, kafka3:9094 --topic test --from-beginning
```

在使用的时候会用到bootstrap与broker.list其实是实现一个功能，broker.list是旧版本命令。

```
bash-4.4# kafka-console-producer.sh --broker-list kafka1:9092,kafka2:9093,kafka3:9094 --topic test --from-beginning
>[2020-10-19 21:59:50,651] WARN [Producer clientId=console-producer] B
NetworkClient)

>lagou hello
>yuanjing hello
>student hello
>^Cbash-4.4# kafka-console-consumer.sh --bootstrap-server kafka1:9092,
yuanjing hello
lagou hello
student hello
```

一、确保消费者消费的消息是顺序的，需要把消息存放在同一个topic的同一个分区

二、一个主题多个分区，分区内的消息有序。

(5) 运行describe的命令

运行describe查看topic的相关详细信息

```
#查看topic主题详情，Zookeeper节点写一个和全部写，效果一致
kafka-topics.sh --describe --zookeeper zoo1:2181,zoo2:2181,zoo3:2181 --topic test

#结果列表
Topic: test1    PartitionCount: 3      ReplicationFactor: 3      Configs:
      Topic: test1    Partition: 0      Leader: 1001      Replicas: 1001,1003,1002      Isr: 1001,1003,1002
      Topic: test1    Partition: 1      Leader: 1002      Replicas: 1002,1001,1003      Isr: 1002,1001,1003
      Topic: test1    Partition: 2      Leader: 1003      Replicas: 1003,1002,1001      Isr: 1003,1002,1001
```

结果说明：

这是输出的解释。第一行给出了所有分区的摘要，每个附加行提供有关一个分区的信息。有几个分区，下面就显示几行

- leader：是负责给定分区的所有读取和写入的节点。每个节点将成为随机选择的分区部分的领导者。
- replicas：显示给定partition所有副本所存储节点的节点列表，不管该节点是否是leader或者是否存活。
- isr：副本都已同步的节点集合，这个集合中的所有节点都是存活状态，并且跟leader同步

(6) 增加topic分区数

任意kafka服务器执行以下命令可以增加topic分区数

```
kafka-topics.sh --zookeeper zkhost:port --alter --topic test --partitions 8
```

(7) 增加配置

flush.messages：此项配置指定时间间隔：强制进行fsync日志，默认值为None。

例如，如果这个选项设置为1，那么每条消息之后都需要进行fsync，如果设置为5，则每5条消息就需要进行一次fsync。

一般来说，建议你不要设置这个值。此参数的设置需要在“数据可靠性”与“性能”之间做必要的权衡。

如果此值过大，将会导致每次“fsync”的时间较长(I/O阻塞)。

如果此值过小，将会导致“fsync”的次数较多，这也意味着整体的client请求有一定的延迟，物理server故障，将会导致没有fsync的消息丢失。

动态修改kafka的配置

```
kafka-topics.sh --zookeeper zoo1:2181 --alter --topic test --config flush.messages=1
```

(8) 删除配置

动态删除kafka集群配置

```
kafka-topics.sh --zookeeper zoo1:2181 --alter --topic test --delete-config flush.messages
```

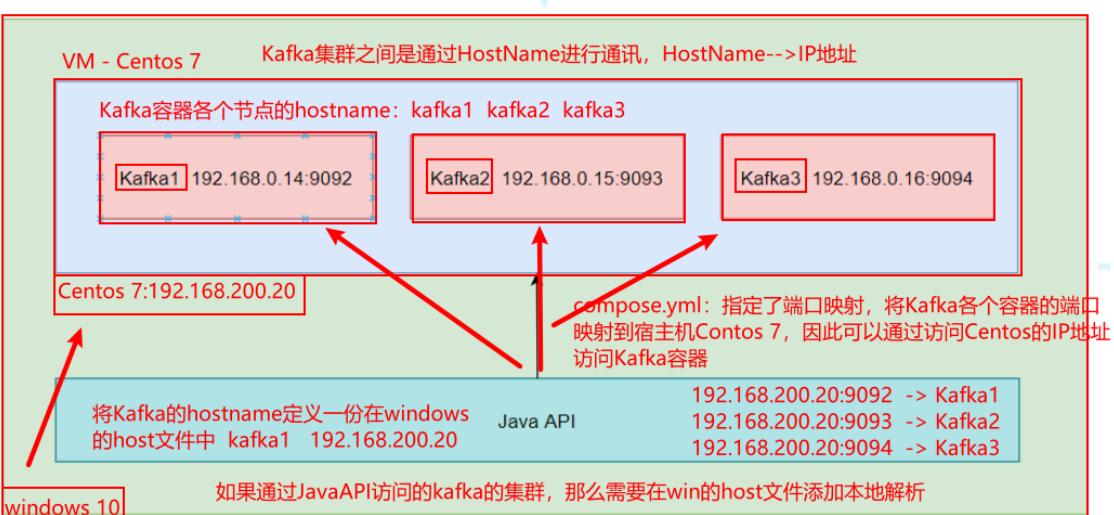
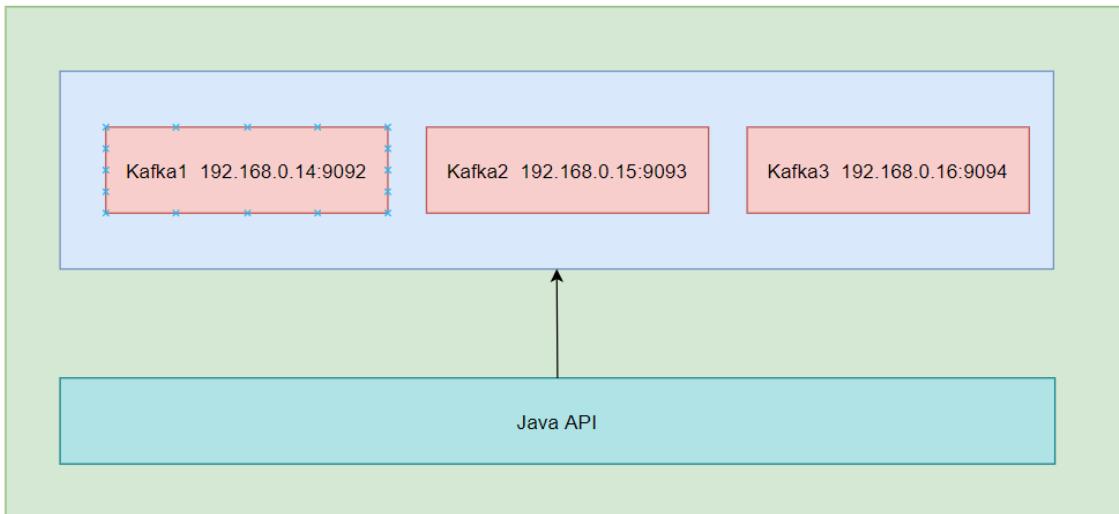
(9) 删除topic

目前删除topic在默认情况只是打上一个删除的标记，在重新启动kafka后才删除。如果需要立即删除，则需要在server.properties中配置：

delete.topic.enable=true (集群中的所有实例节点)，一个主题会在不同的kafka节点中分配分组信息和副本信息
然后执行以下命令进行删除topic

```
kafka-topics.sh --zookeeper zoo1:2181 --delete --topic test
```

6. Java API操作kafka



修改Windows的Host文件:

目录: C:\Windows\System32\drivers\etc (win10)

内容:

```
192.168.200.20 kafka1
192.168.200.20 kafka2
192.168.200.20 kafka3
```

- 创建maven的工程, 导入kafka相关的依赖

[kafka_2.11-1.0.0.tgz](#)

```
<dependencies>
    <!-- https://mvnrepository.com/artifact/org.apache.kafka/kafka-clients -->
    <dependency>
        <groupId>org.apache.kafka</groupId>
        <artifactId>kafka-clients</artifactId>
        <version>1.0.0</version>
    </dependency>
</dependencies>

<build>
    <plugins>
        <!-- java编译插件 -->
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.2</version>
            <configuration>
                <source>1.8</source>
                <target>1.8</target>
                <encoding>UTF-8</encoding>
            </configuration>
        </plugin>
    </plugins>
</build>
```

```
</plugin>
</plugins>
</build>
```

6.1 生产者代码

```
public class ProducerDemo {

    public static String topic = "lagou";//定义主题

    public static void main(String[] args) throws Exception {
        Properties properties = new Properties();
        properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
        "192.168.200.20:9092,192.168.200.20:9093,192.168.200.20:9094");
        //网络传输,对key和value进行序列化
        properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
        properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
        //创建消息生产对象,需要从properties对象或者从properties文件中加载信息
        KafkaProducer<String, String> kafkaProducer = new KafkaProducer<>(properties);

        try {
            while (true) {
                //设置消息内容
                String msg = "Hello," + new Random().nextInt(100);
                //将消息内容封装到ProducerRecord中
                ProducerRecord<String, String> record = new ProducerRecord<String, String>(topic, msg);
                kafkaProducer.send(record);
                System.out.println("消息发送成功:" + msg);
                Thread.sleep(500);
            }
        } finally {
            kafkaProducer.close();
        }
    }
}
```

6.2 消费者代码

```
public class ConsumerDemo {

    public static void main(String[] args){
        Properties p = new Properties();
        p.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
        "192.168.200.20:9092,192.168.200.20:9093,192.168.200.20:9094");
        p.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
        p.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
        //指定组名
        p.put(ConsumerConfig.GROUP_ID_CONFIG, "lagou");

        KafkaConsumer<String, String> kafkaConsumer = new KafkaConsumer<String, String>(p);
        kafkaConsumer.subscribe(Collections.singletonList(ProducerDemo.topic));// 订阅消息

        while (true) {
            ConsumerRecords<String, String> records = kafkaConsumer.poll(100);
            for (ConsumerRecord<String, String> record : records) {
                System.out.println(String.format("topic:%s,offset:%d,消息:%s", // record.topic(), record.offset(), record.value()));
            }
        }
    }
}
```

7. Apache kafka原理

7.1 分区副本机制

kafka有三层结构：kafka有多个主题，每个主题有多个分区，每个分区又有多条消息。

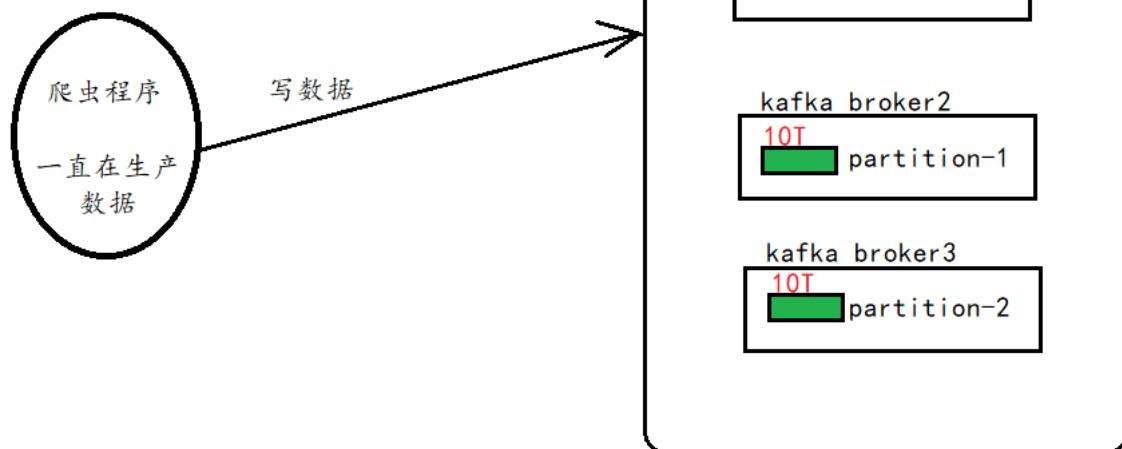
分区机制：主要解决了单台服务器存储容量有限和单台服务器并发数限制的问题 一个分片的不同副本不能放到同一个broker上。

当主题数据量非常大的时候，一个服务器存放不了，就将数据分成两个或者多个部分，存放在多台服务器上。每个服务器上的数据，叫做一个分片

假设：爬虫程序一天向kafka发送10T消息

topic:newsJson: 最开始想其中一台写数据，每天10T

经过3天后，就会变为30T数据，但是一个服务器可能只有24T数据，一台放不下，进行分片存储



分区对于 Kafka 集群的好处是：实现负载均衡，高存储能力、高伸缩性。分区对于消费者来说，可以提高并发度，提高效率。

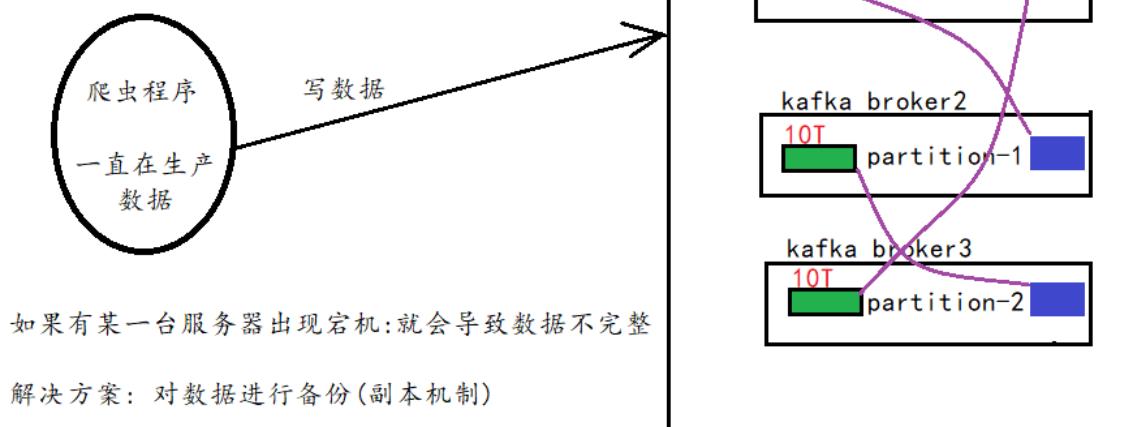
副本：副本备份机制解决了数据存储的高可用问题

当数据只保存一份的时候，有丢失的风险。为了更好的容错和容灾，将数据拷贝几份，保存到不同的机器上。

假设：爬虫程序一天向kafka发送10T消息

topic:newsJson: 最开始想其中一台写数据，每天10T

经过3天后，就会变为30T数据，但是一个服务器可能只有24T数据，一台放不下，进行分片存储



多个follower副本通常存放在和leader副本不同的broker中。通过这样的机制实现了高可用，当某台机器挂掉后，其他follower副本也能迅速“转正”，开始对外提供服务。

kafka的副本都有哪些作用？

在kafka中，实现副本的目的就是冗余备份，且仅仅是冗余备份，所有的读写请求都是由leader副本进行处理的。follower副本仅有一个功能，那就是从leader副本拉取消息，尽量让自己跟leader副本的内容一致。

说说follower副本为什么不对外提供服务？

这个问题本质上是对性能和一致性的取舍。试想一下，如果follower副本也对外提供服务那会怎么样呢？首先，性能是肯定会有所提升的。但同时，会出现一系列问题。类似数据库事务中的幻读，脏读。

比如你现在写入一条数据到kafka主题a，消费者b从主题a消费数据，却发现消费不到，因为消费者b去读取的那个分区副本中，最新消息还没写入。而这个时候，另一个消费者c却可以消费到最新那条数据，因为它消费了leader副本。

为了提高那么些性能而导致出现数据不一致问题，那显然是不值得的。

7.2 kafka保证数据不丢失机制

从Kafka的大体角度上可以分为数据生产者，Kafka集群，还有就是消费者，而要保证数据的不丢失也要从这三个角度去考虑。

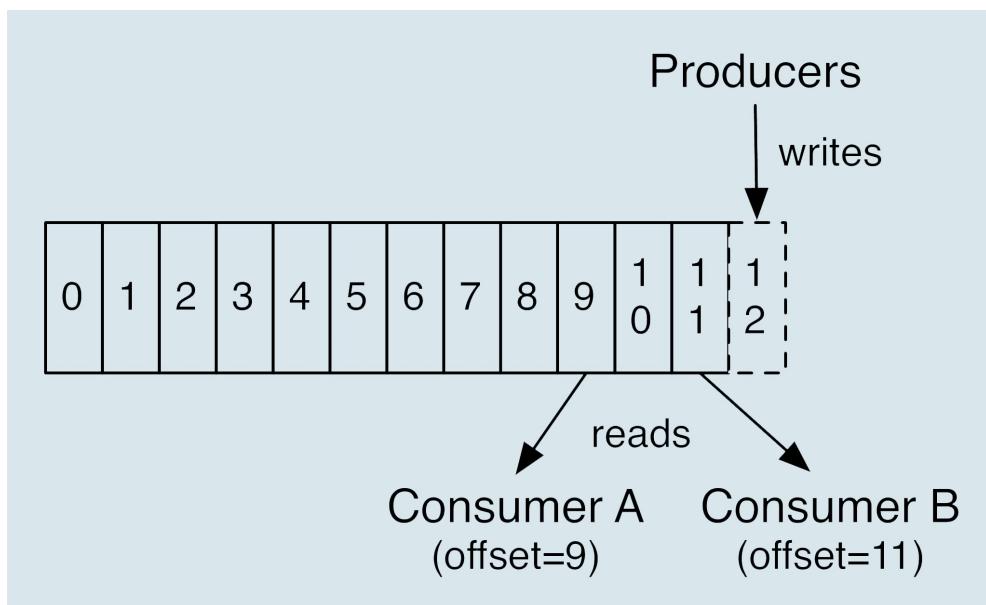
7.2.1. 消息生产者

消息生产者保证数据不丢失：消息确认机制（ACK机制），参考值有三个：0,1, -1

```
//producer无需等待来自broker的确认而继续发送下一批消息。  
//这种情况下数据传输效率最高，但是数据可靠性确是最低的。  
properties.put(ProducerConfig.ACKS_CONFIG, "0");  
  
//producer只要收到一个分区副本成功写入的通知就认为推送消息成功了。  
//这里有一个地方需要注意，这个副本必须是leader副本。  
//只有leader副本成功写入了，producer才会认为消息发送成功。  
properties.put(ProducerConfig.ACKS_CONFIG, "1");  
  
//ack=-1，简单来说就是，producer只有收到分区内所有副本的成功写入的通知才认为推送消息成功。  
properties.put(ProducerConfig.ACKS_CONFIG, "-1");
```

7.2.2 消息消费者

kafka消费消息的模型：



即消费消息，设置好offset，类比一下：

Kafka动作	看书动作
消费消息	看书
offset位移	书签

什么时候消费者丢失数据呢？

由于Kafka consumer默认是自动提交位移的（先更新位移，再消费消息），如果消费程序出现故障，没消费完毕，则丢失了消息，此时，broker并不知道。

解决方案：

enable.auto.commit=false 关闭自动提交位移

在消息被完整处理之后再手动提交位移

```
properties.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "false");
```

7.3 消息存储及查询机制

kafka 使用日志文件的方式来保存生产者消息，每条消息都有一个 offset 值来表示它在分区中的偏移量。

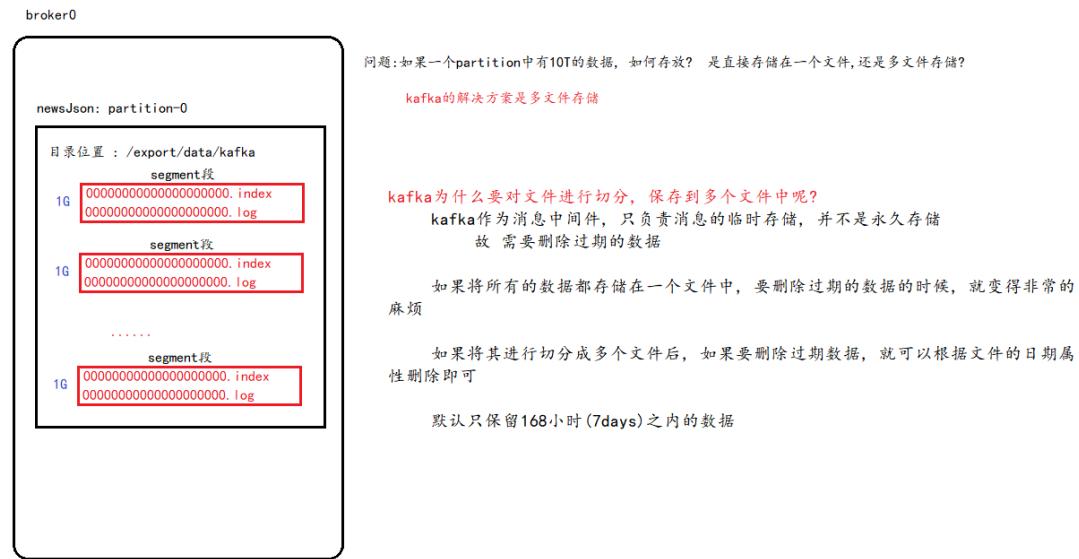
Kafka 中存储的一般都是海量的消息数据，为了避免日志文件过大，一个分片 并不是直接对应在一个磁盘上的日志文件，而是对应磁盘上的一个目录，这个目录的命名规则是<topic_name>_<partition_id>。

kafka容器数据目录：

/kafka/kafka-logs-kafka1

```
bash-4.4# ls -a
.
..
.lock
_consumer_offsets-1
_consumer_offsets-10
_consumer_offsets-13
_consumer_offsets-16
_consumer_offsets-19
_consumer_offsets-22
_consumer_offsets-25
_consumer_offsets-28
_consumer_offsets-31
_consumer_offsets-34
_consumer_offsets-37
_consumer_offsets-4
_consumer_offsets-40
_consumer_offsets-43
_consumer_offsets-46
_consumer_offsets-49
_consumer_offsets-7
cleaner-offset-checkpoint
lagou-0
log-start-offset-checkpoint
meta.properties
recovery-point-offset-checkpoint
replication-offset-checkpoint
```

7.3.1 消息存储机制



log分段：

每个分片目录中，kafka 通过分段的方式将 数据分为多个 LogSegment。

一个 LogSegment 对应磁盘上的一个日志文件 (00000000000000000000.log) 和一个索引文件(如上：
00000000000000000000.index)。

其中日志文件是用来记录消息的。索引文件是用来保存消息的索引。

每个LogSegment 的大小可以在server.properties 中log.segment.bytes=107370 (设置分段大小,默认是1gb) 选项进行设置。

当log文件等于1G时，新的会写入到下一个segment中。

```
| 00000000000000000000.index      00000000000000000000.log      00000000000000000000.timeindex  leader-epoch-checkpoint
  timeindex文件，是kafka的具体时间日志
```

7.3.2 通过 offset 查找 message

存储的结构：一个主题 --> 多个分区 ----> 多个日志段（多个文件）

第一步：查询segment file：

segment file命名规则跟offset有关，根据segment file可以知道它的起始偏移量，因为Segment file的命名规则是上一个segment文件最后一条消息的offset值。所以只要根据offset 二分查找文件列表，就可以快速定位到具体文件。

比如

第一个segment file是00000000000000000000.index表示最开始的文件，起始偏移量(offset)为0。

第二个是00000000000000091932.index：代表消息量起始偏移量为91933 = 91932 + 1。那么offset=5000时应该定位
00000000000000000000.index

第二步通过segment file查找message：

通过第一步定位到segment file，当offset=5000时，依次定位到00000000000000000000.index的元数据物理位置和
00000000000000000000.log的物理偏移地址，然后再通过00000000000000000000.log顺序查找直到offset=5000为止。

7.4 生产者消息分发策略

kafka在数据生产的时候，有一个数据分发策略。默认的情况使用DefaultPartitioner.class类。

这个类中就定义数据分发的策略。

```
public interface Partitioner extends Configurable, Closeable {  
  
    /**  
     * Compute the partition for the given record.  
     *  
     * @param topic The topic name  
     * @param key The key to partition on (or null if no key)  
     * @param keyBytes The serialized key to partition on( or null if no key)  
     * @param value The value to partition on or null  
     * @param valueBytes The serialized value to partition on or null  
     * @param cluster The current cluster metadata  
     */  
    public int partition(String topic, Object key, byte[] keyBytes, Object value, byte[] valueBytes, Cluster  
cluster);  
  
    /**  
     * This is called when partitioner is closed.  
     */  
    public void close();  
}
```

默认实现类：org.apache.kafka.clients.producer.internals.DefaultPartitioner

1) 如果是用户指定了partition，生产就不会调用DefaultPartitioner.partition()方法

数据分发策略的时候，可以指定数据发往哪个partition。

当ProducerRecord 的构造参数中有partition的时候，就可以发送到对应partition上

```
/**  
 * Creates a record to be sent to a specified topic and partition  
 *  
 * @param topic The topic the record will be appended to  
 * @param partition The partition to which the record should be sent  
 * @param key The key that will be included in the record  
 * @param value The record contents  
 */  
public ProducerRecord<String, Integer> partition(String topic, Integer partition, K key, V value) {  
    this(topic, partition, null, key, value, null);  
}
```

2) DefaultPartitioner源码

如果指定key，是取决于key的hash值

如果不指定key，轮询分发

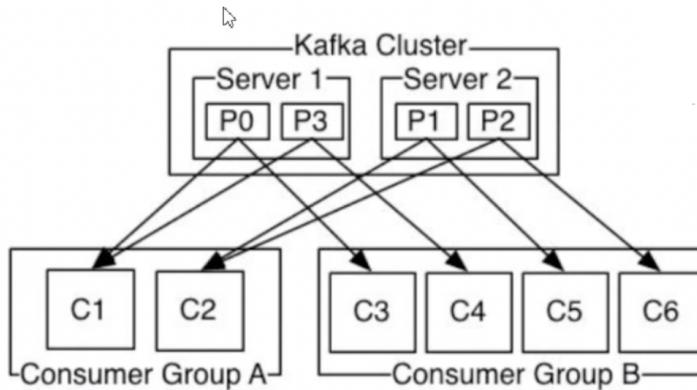
```
public int partition(String topic, Object key, byte[] keyBytes, Object value, byte[] valueBytes, Cluster  
cluster) {  
    //获取该topic的分区列表  
    List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);  
    //获得分区的个数  
    int numPartitions = partitions.size();  
    //如果key值为null  
    if (keyBytes == null) {//如果没有指定key，那么就是轮询  
        //维护一个key为topic的ConcurrentHashMap，并通过CAS操作的方式对value值执行递增+1操作  
        int nextValue = nextValue(topic);  
        //获取该topic的可用分区列表  
        List<PartitionInfo> availablePartitions = cluster.availablePartitionsForTopic(topic);  
        if (availablePartitions.size() > 0) {//如果可用分区大于0  
            //执行求余操作，保证消息落在可用分区上  
            int part = Utils.toPositive(nextValue) % availablePartitions.size();  
            return availablePartitions.get(part).partition();  
        } else {  
            // 没有可用分区的话，就给出一个不可用分区  
            return Utils.toPositive(nextValue) % numPartitions;  
        }  
    } else {//不过指定了key，key肯定就不为null  
        // 通过计算key的hash，确定消息分区  
        return Utils.toPositive(Utils.murmur2(keyBytes)) % numPartitions;  
    }
```

```
}
```

7.5 消费者负载均衡机制

同一个分区中的数据，只能被一个消费者组中的一个消费者所消费。例如 P0分区中的数据不能被Consumer Group A中C1与C2同时消费。

消费组：一个消费组中可以包含多个消费者，`properties.put(ConsumerConfig.GROUP_ID_CONFIG,"groupName")`；如果该消费组有四个消费者，主题有四个分区，那么每人一个。多个消费组可以重复消费消息。



<https://blog.csdn.net/u010899985>

- 如果有3个Partition, p0/p1/p2, 同一个消费组有3个消费者, c0/c1/c2, 则为一一对应关系;
- 如果有3个Partition, p0/p1/p2, 同一个消费组有2个消费者, c0/c1, 则其中一个消费者消费2个分区的数据, 另一个消费者消费一个分区的数据;
- 如果有2个Partition, p0/p1, 同一个消费组有3个消费者, c0/c1/c3, 则其中有一个消费者空闲, 另外2个消费者消费分别各自消费一个分区的数据;

8. kafka配置文件说明

server.properties

1、broker.id=0:

kafka集群是由多个节点组成的，每个节点称为一个broker，中文翻译是代理。每个broker都有一个不同的brokerId，由`broker.id`指定，是一个不小于0的整数，各brokerId必须不同，但不必连续。如果我们想扩展kafka集群，只需引入新节点，分配一个不同的`broker.id`即可。

启动kafka集群时，每一个broker都会实例化并启动一个kafkaController，并将该broker的`brokerId`注册到zooKeeper的相应节点中。集群各broker会根据选举机制选出其中一个broker作为leader，即leader kafkaController。leader kafkaController负责主题的创建与删除、分区和副本的管理等。当leader kafkaController宕机后，其他broker会再次选举出新的leader kafkaController。

2、log.dir = /export/data/kafka/

broker持久化消息到哪里，数据目录

3、log.retention.hours = 168

log文件最小存活时间，默认是168h，即7天。相同作用的还有`log.retention.minutes`、`log.retention.ms`。`retention`是保存的意思。

数据存储的最大时间超过这个时间会根据`log.cleanup.policy`设置的策略处理数据，也就是消费端能够多久去消费数据。

`log.retention.bytes`和`log.retention.hours`任意一个达到要求，都会执行删除，会被topic创建时的指定参数覆盖。

4、log.retention.check.interval.ms

多长时间检查一次是否有log文件要删除。默认是300000ms，即5分钟。

5、log.retention.bytes

限制单个分区的log文件的最大值，超过这个值，将删除旧的log，以满足log文件不超过这个值。默认是-1，即不限制。

6、log.roll.hours

多少时间会生成一个新的log segment， 默认是168h， 即7天。相同作用的还有log.roll.ms、segment.ms。

7、log.segment.bytes

log segment多大之后会生成一个新的log segment， 默认是1073741824， 即1G。

8、log.flush.interval.messages

指定broker每收到几个消息就把消息从内存刷到硬盘（刷盘）。默认是9223372036854775807 好大。

kafka官方不建议使用这个配置，建议使用副本机制和操作系统的后台刷新功能，因为这更高效。这个配置可以根据不同的topic设置不同的值，即在创建topic的时候设置值。

补充说明：

在Linux操作系统中，当我们把数据写入到文件系统之后，数据其实在操作系统的page cache里面，并没有刷到磁盘上去。如果此时操作系统挂了，其实数据就丢了。

1、kafka是多副本的，当你配置了同步复制之后。多个副本的数据都在page cache里面，出现多个副本同时挂掉的概率比1个副本挂掉，概率就小很多了

2、操作系统有后台线程，定期刷盘。如果应用程序每写入1次数据，都调用一次fsync，那性能损耗就很大，所以一般都会在性能和可靠性之间进行权衡。因为对应一个应用来说，虽然应用挂了，只要操作系统不挂，数据就不会丢。

9、log.flush.interval.ms

指定broker每隔多少毫秒就把消息从内存刷到硬盘。默认值同log.flush.interval.messages一样， 9223372036854775807。

同log.flush.interval.messages一样， kafka官方不建议使用这个配置。

10、delete.topic.enable=true

是否允许从物理上删除topic

9. kafka监控与运维

9.1 kafka-eagle概述

在生产环境下，在Kafka集群中，消息数据变化是我们关注的问题，当业务前提不复杂时，我们可以使用Kafka命令提供带有Zookeeper客户端工具的工具，可以轻松完成我们的工作。随着业务的复杂性，增加Group和Topic，那么我们使用Kafka提供命令工具，已经感到无能为力，那么Kafka监控系统目前尤为重要，我们需要观察消费者应用的细节。

为了简化开发者和服务工程师维护Kafka集群的工作有一个监控管理工具，叫做 Kafka-eagle。这个管理工具可以很容易地发现分布在集群中的哪些topic分布不均匀，或者是分区在整个集群分布不均匀的情况。它支持管理多个集群、选择副本、副本重新分配以及创建Topic。同时，这个管理工具也是一个非常好的可以快速浏览这个集群的工具，

9.2 搭建安装 kafka-eagle

环境要求:需要安装jdk，启动zk以及kafka的服务

```
# 启动Zookeeper
zkServer.sh start

#启动Kafka
nohup ./kafka-server-start.sh /export/servers/kafka/config/server.properties 2>&1 &
```

修改windows host文件

```
127.0.0.1      LagouCloudEurekaServerB  
127.0.0.1      LagouCloudEurekaServerA  
127.0.0.1      localhost  
127.0.0.1      www.xmind.net  
192.168.200.20 kafka1  
192.168.200.20 kafka2  
192.168.200.20 kafka3  
192.168.200.11 node1  
192.168.200.12 node2  
192.168.200.13 node3
```

搭建步骤:

- 1) 下载kafka-eagle的源码包

kafka-eagle官网：

<http://download.kafka-eagle.org/>

我们可以从官网上面直接下载最新的安装包即可kafka-eagle-bin-1.3.2.tar.gz这个版本即可

代码托管地址:

<https://github.com/smartloli/kafka-eagle/releases>

- 2) 上传安装包并解压;

这里我们选择将kafka-eagle安装在第三台

如果要解压的是zip格式，需要先安装命令支持。

- yum install unzip
 - unzip xxxx.zip

```
#将安装包上传至 node01服务器的/export/softwares路径下，然后解压  
cd /export/softwares/  
unzip kafka-eagle.zip  
cd cd kafka-eagle-web/target/  
tar -zxf kafka-eagle-web-2.0.1-bin.tar.gz -C /export/servers
```

- 3) 准备数据库:

kafka-eagle需要使用一个数据库来保存一些元数据信息，我们这里直接使用mysql数据库来保存即可，在node01服务器执行以下命令创建一个mysql数据库即可。

SOLite, MySQL

--进入mysql客户端：
create database eagle;

- 4) 修改kafka-eagle配置文件

```
#####
# multi zookeeper & kafka cluster list
#####
kafka.eagle.zk.cluster.alias=cluster1,cluster2 指定集群的别名，可以有多个集群
cluster1.zk.list=tdn1:2181,tdn2:2181,tdn3:2181
cluster2.zk.list=xnd10:2181,xnd11:2181,xnd12:2181
#####

```

```
cd /export/servers/kafka-eagle-bin-1.3.2/kafka-eagle-web-1.3.2/conf  
vim system-config.properties  
  
#内容如下：  
kafka.eagle.zk.cluster.alias=cluster1  
cluster1.zk.list=node1:2181,node2:2181,node3:2181  
  
kafka.eagle.driver=com.mysql.jdbc.Driver  
kafka.eagle.url=jdbc:mysql://10.1.192.208:3306/eagle  
kafka.eagle.username=root  
kafka.eagle.password=wu7787879
```

默认情况下MySQL只允许本机连接到MySQL实例中，所以如果要远程访问，必须开放权限：

```
update user set host = '%' where user = 'root'; //修改权限
```

flush privileges; //刷新配置

- ## ● 5) 配置环境变量

kafka-eagle必须配置环境变量 node03服务器执行以下命令来进行配置环境变量

```
vi /etc/profile
#内容如下:
export KE_HOME=/export/servers/kafka-eagle-bin-1.3.2/kafka-eagle-web-1.3.2
export PATH=$KE_HOME/bin:$PATH

#让修改立即生效, 执行
source /etc/profile
```

- 6) 启动kakfa-eagle

```
cd kafka-eagle-web-1.3.2/bin
chmod u+x ke.sh
./ke.sh start
```

- 6) 访问主界面:

<http://node03:8048/ke/account/signin?/ke/>

用户名: admin

密码: 123456

