

CONTENTS



1 介绍




2 架构



3 搭建




4 Java API使用



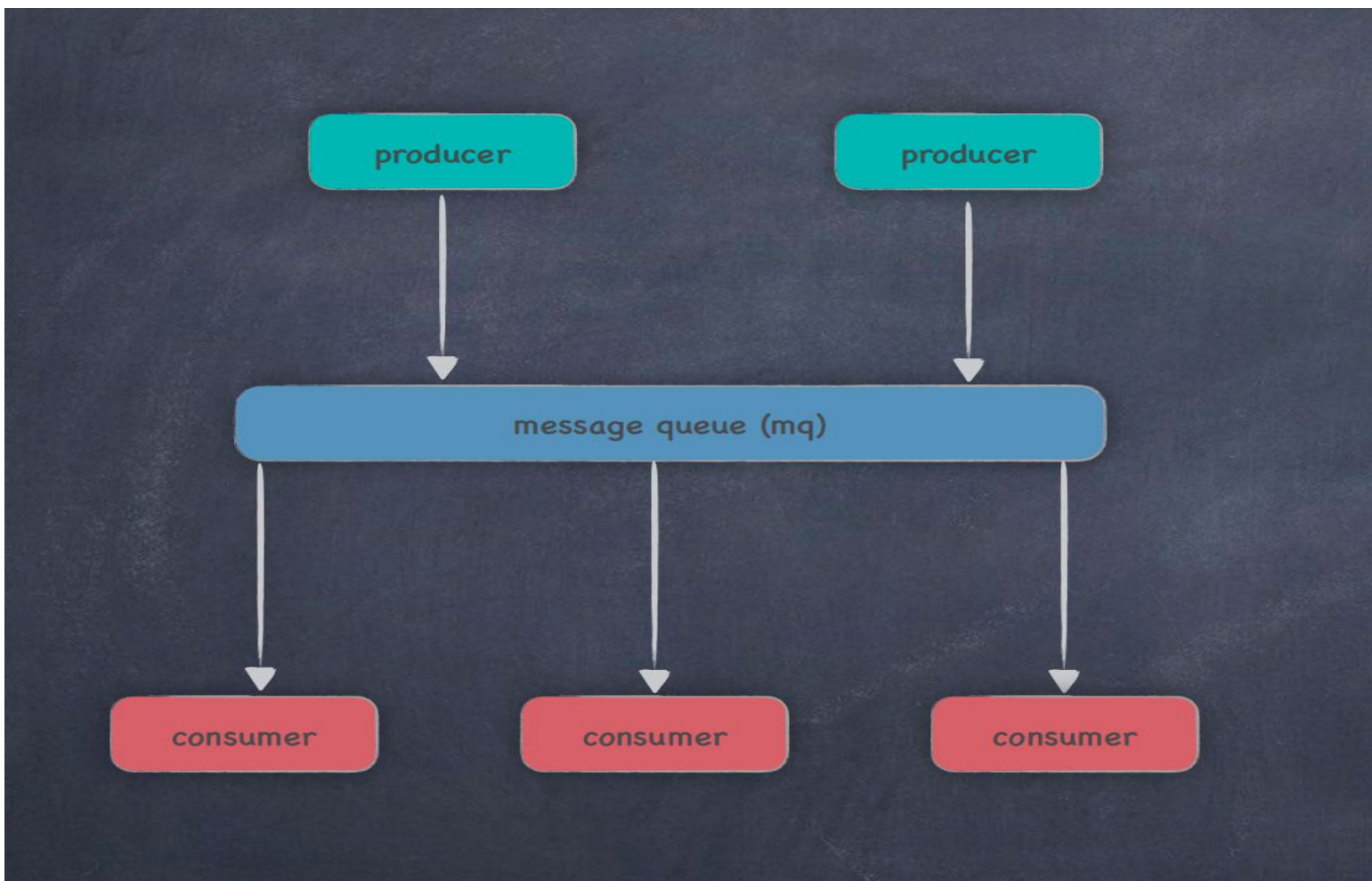
PART 01

介绍



什么是Kafka

Kafka是由Apache开源，使用scala和java语言编写的一个分布式的基于发布/订阅模式的消息队列。





推送VS拉取

推送(push):

push模式下消息的**实时性更高**，对于消费者使用来说更简单，反正有消息来了就会推过来。

但push模式很难适应消费速率不同的消费者，因为消息发送速率是由服务端决定的。push模式的目标是尽可能以最快速度传递消息，但是这样很容易造成消费者来不及处理消息，典型的表现就是拒绝服务以及网络拥塞。

拉取(pull):

pull模式主动权就在消费者身上了，消费者可以根据自身的情况来发起拉取消息的请求。假设当前消费者觉得自己消费不过来了，它可以根据一定的策略停止拉取，或者间隔拉取都行。

消息延迟，因为是消费者是主动拉取消息，它并不知道消息的存在与否，于是只能不停的去拉取，但又不能很频繁，太频繁对服务端的压力也很大。

Kafka的解决方案:

首先Kafka采用了pull拉取消息的模式，消费者采用**长轮询**的方式去服务端拉取消息时定义了一个超时时间，如果有马上返回消息，没有的话消费者等着直到超时，然后再次发起拉消息的请求。



消息队列解决了什么问题

解耦合：

当一个模块的处理需要强依赖另一个模块时，假如被依赖的模块出现故障，自然也会影响调用模块的运行，引入消息队列后，可以先将请求参数保存至消息队列中，被依赖模块从消息队列中获取请求参数后执行。

异步操作：

假如一个服务调用另一个服务时间比较长时，可以使用消息队列进行异步处理。一些没有强制关联的操作，也可以使用消息队列来加快响应速度，比如注册时的发送邮箱确认，注册后可以立即给出响应，发邮件的操作可以交给消息队列去做。

流量削峰：

比如秒杀下订单的时候会有大量用户在同一时间请求服务，可能会导致服务因此宕机，此时便可以引入消息队列，对请求进行排队和限制，然后进行慢慢消化。



为什么选择Kafka

| 特性 | ActiveMQ | RabbitMQ | Kafka | RocketMQ |
|---------|----------|------------------------|----------|------------|
| 所属社区/公司 | Apache | Mozilla Public License | Apache | Apache/Ali |
| API完备性 | 高 | 高 | 高 | 低（静态配置） |
| 多语言支持 | 支持JAVA优先 | 语言无关 | 支持JAVA优先 | 支持 |
| 单机吞吐量 | 万级（最差） | 万级 | 十万级 | 十万级（最高） |
| 消息延迟 | - | 微秒级 | 毫秒级 | - |
| 可用性 | 高（主从） | 高（主从） | 非常高（分布式） | 高 |
| 消息丢失 | - | 低 | 理论上不会丢失 | - |
| 消息重复 | - | 可控制 | 理论上会有重复 | - |
| 事务 | 支持 | 不支持 | 支持 | 支持 |
| 文档的完备性 | 高 | 高 | 高 | 中 |

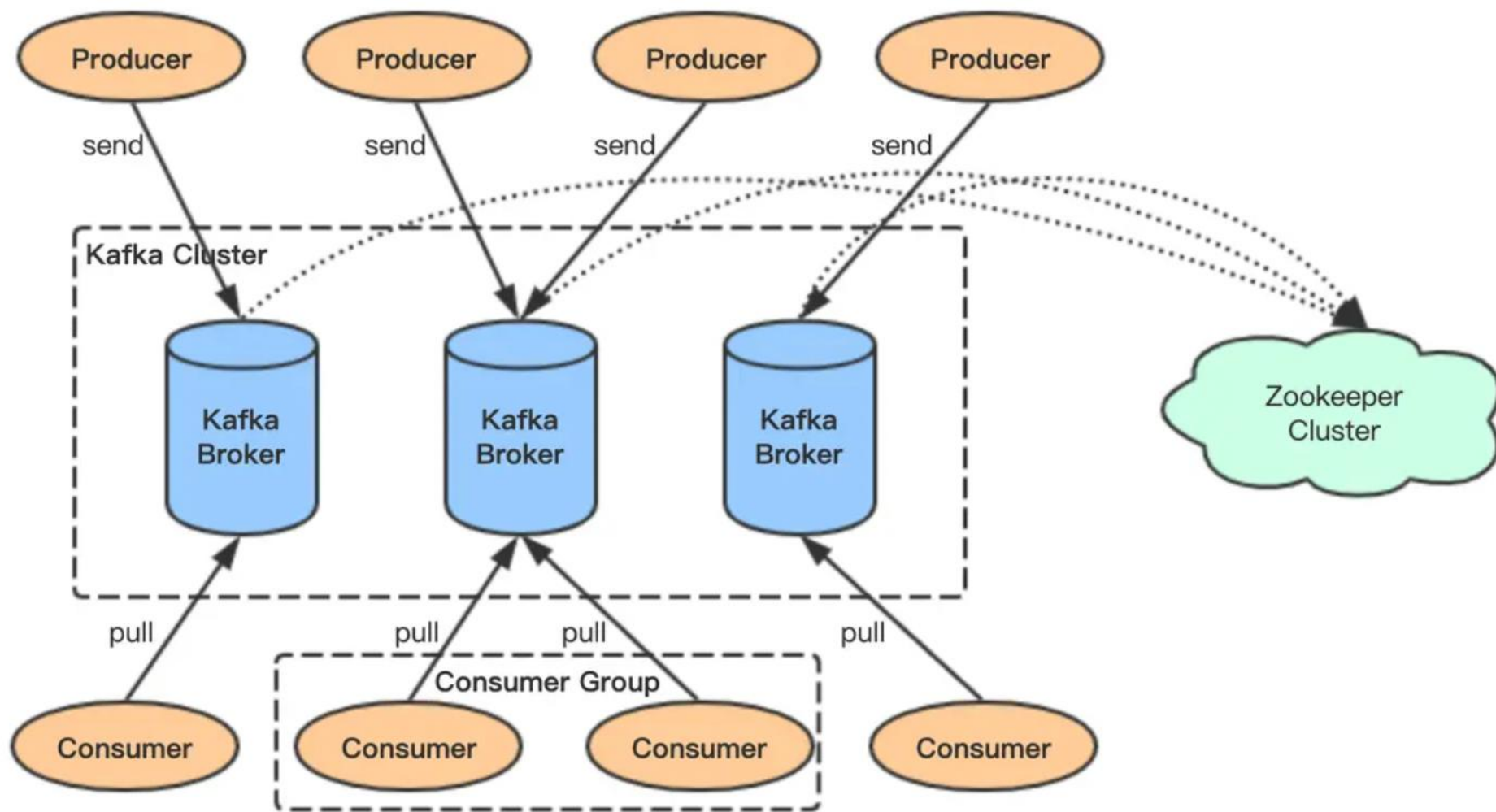


PART 02

架构

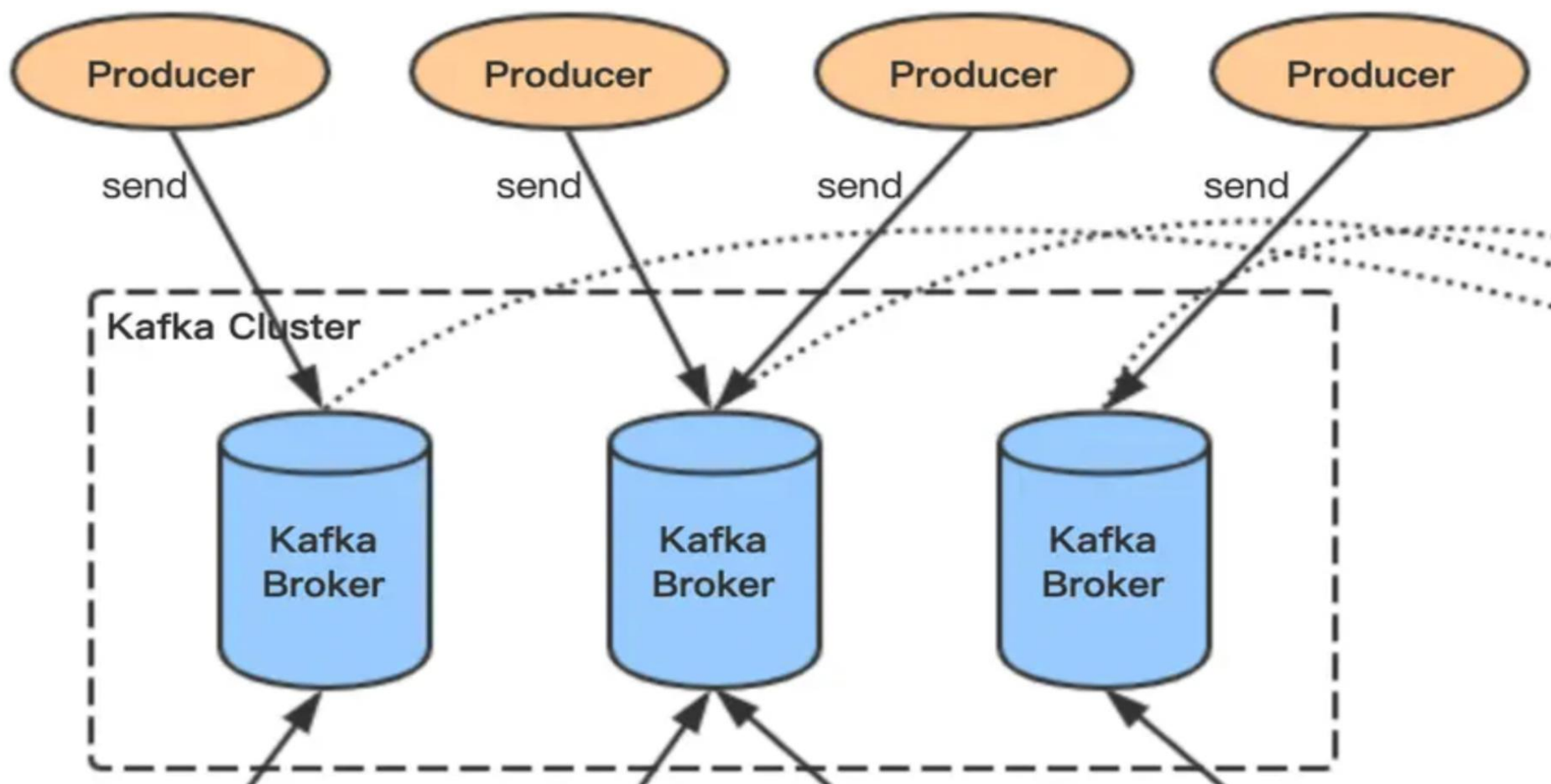


架构图



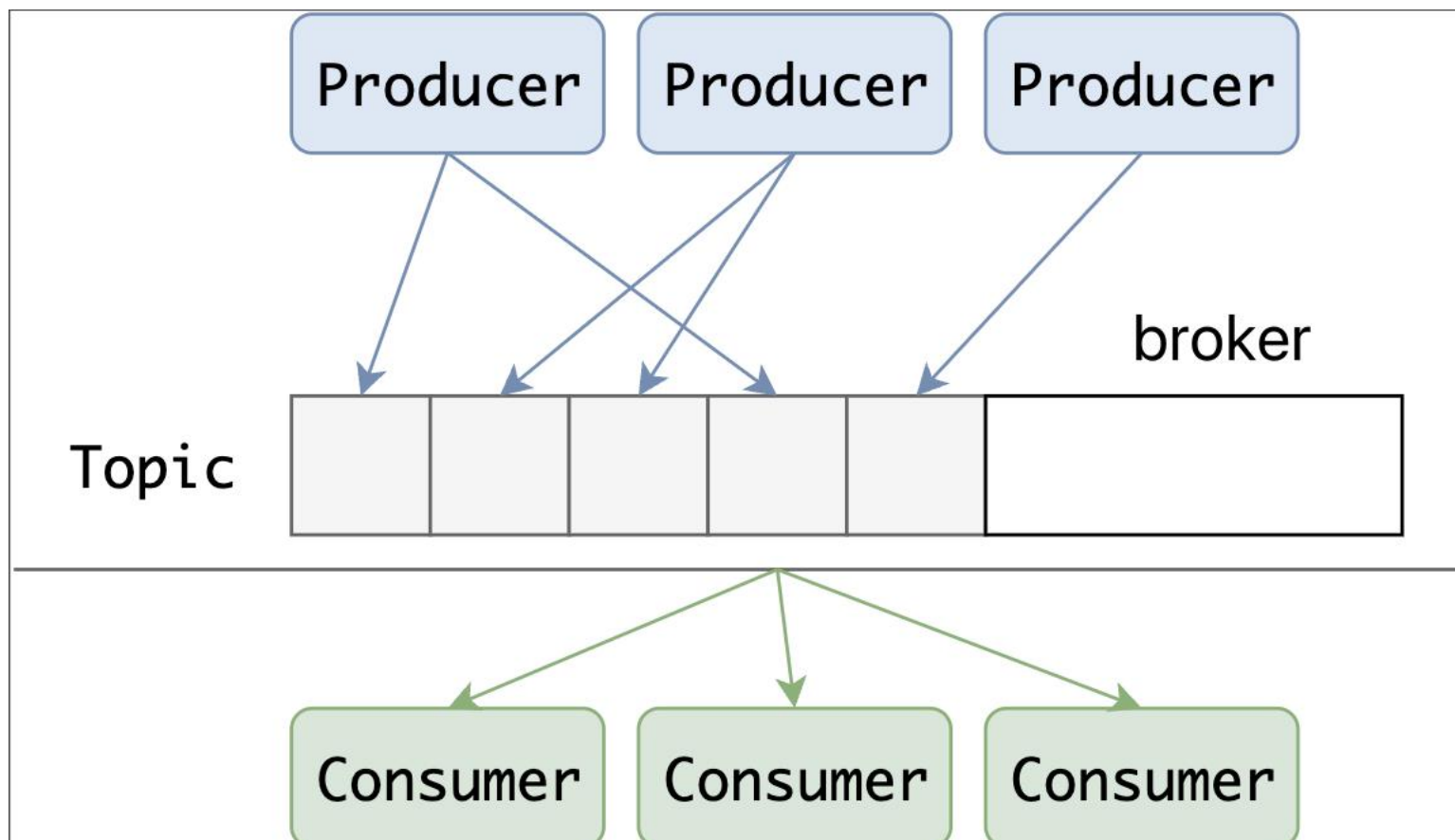
Broker

Broker: 已发布消息保存在一组服务器中，称之为Kafka集群。集群中的每一个服务器都是一个代理(Broker)，反言之每一个Broker就是一个Kafka的实例。



Topic

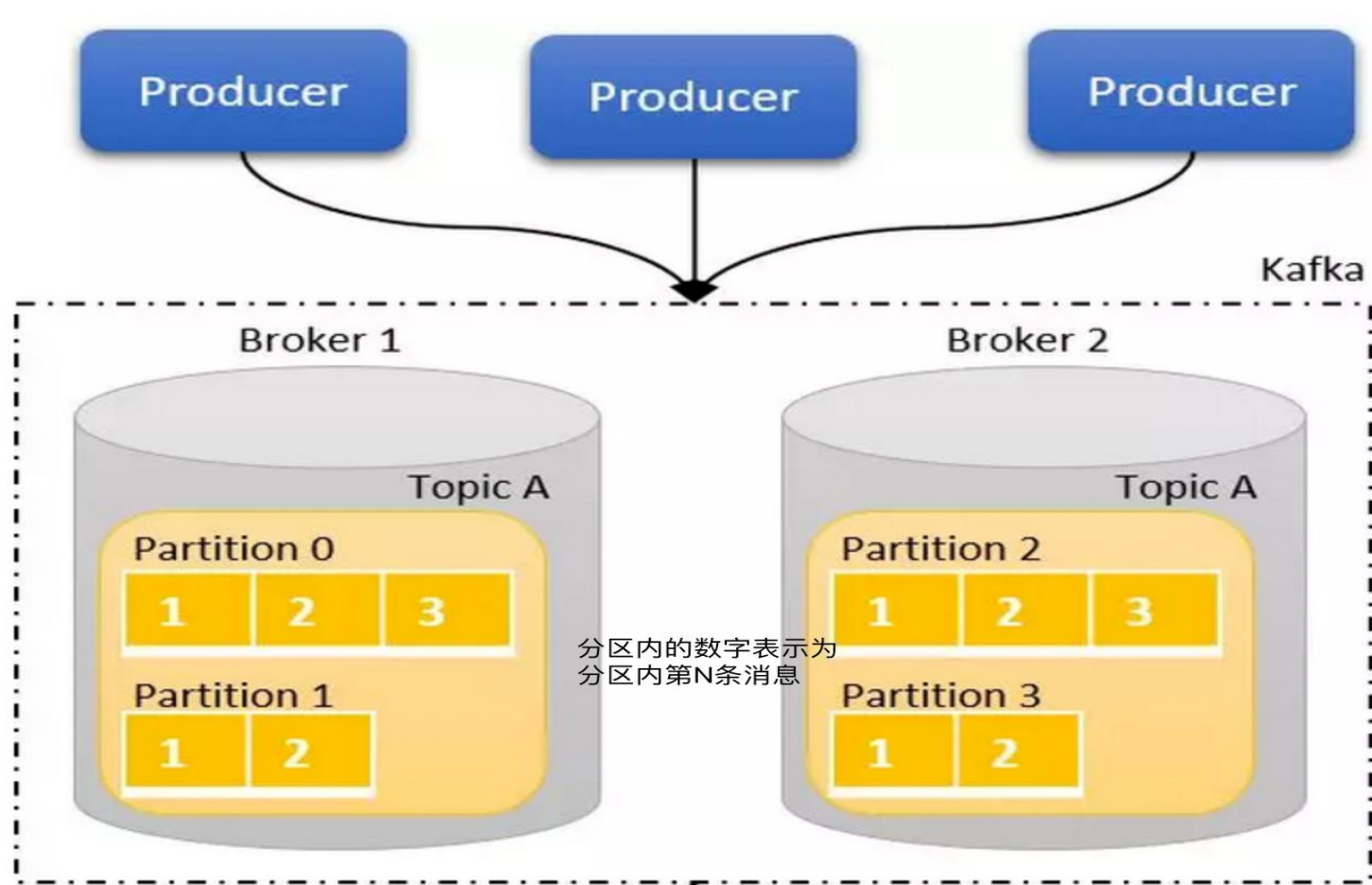
Topic: 是kafka下消息的类别，每一类消息都可以称为一个topic，是逻辑上的概念，用来区分、隔离不同的消息，开发时大多只需要关注消息存到了哪个topic，需要从哪个topic中取数据即可。



Kafka支持多生产者
和多消费者

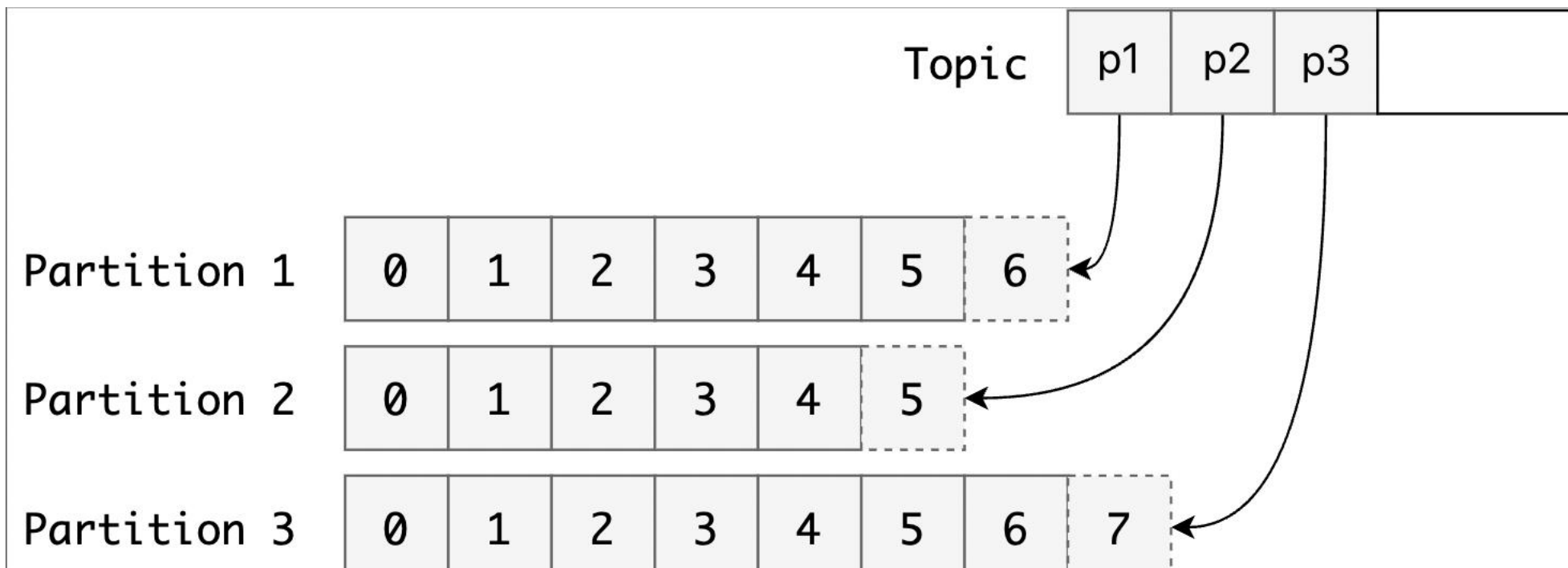
Partition

Partition: 分区，为了负载均衡，将同一个Topic分为多个分区分别存储在集群中不同的broker上，每一个Topic可以有多个分区，但一个分区只属于一个主题，同一个主题中不同分区之间的数据不一样。这样做的好处是当存储不足时可以通过增加分区实现横向扩展。分区数也不是越多越好，配置为broker节点的整数倍即可。



Offset

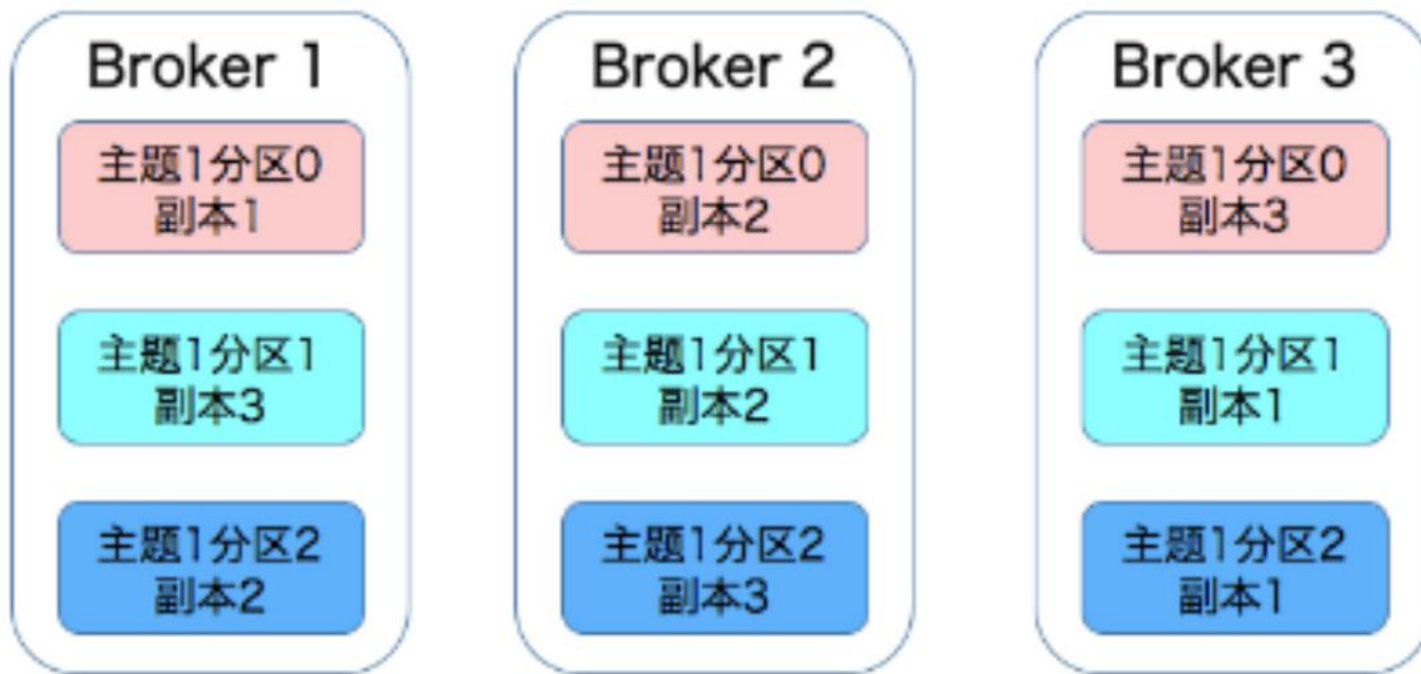
Offset: 写入topic的消息会被平均分配到每一个分区，到达分区后会落实到一个文件中，消息在文件的位置就是offset(偏移量)，是消息在分区中的唯一标识。（开发时可以灵活的使用该属性进行消息消费，可以指定消费者从指定的offset开始消费）



Replicas

Replicas: 分区的副本，保障了分区的高可用，每个分区在每个broker上最多只有一个副本。

下图代表topic1有三个分区，每个分区有三个副本，并且散落在不同的broker上





Consumer Group

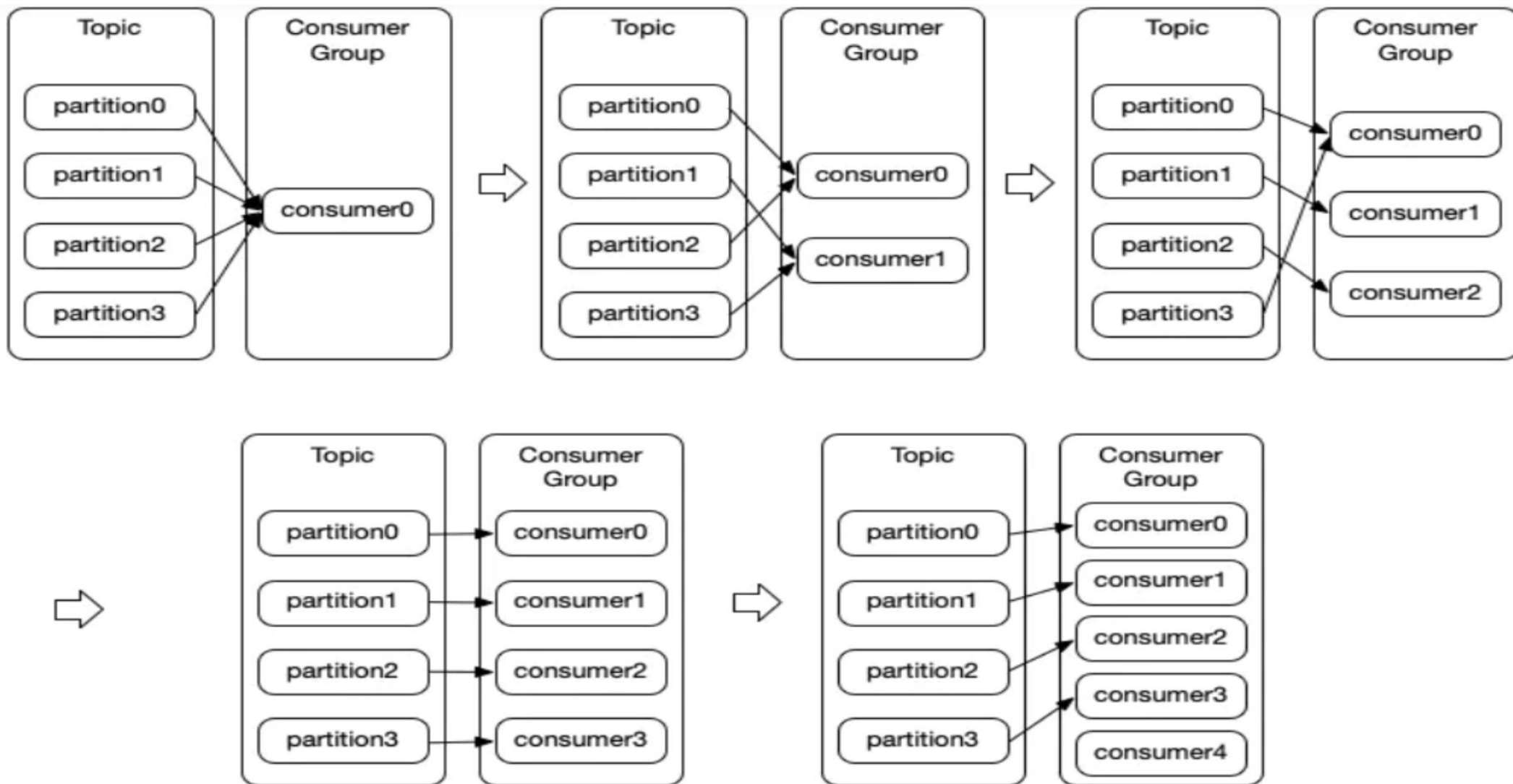
消费者组：

当生产者向 Topic 写入消息的速度超过了消费者（consumer）的处理速度，导致大量的消息在 Kafka 中淤积，此时需要对消费者进行横向伸缩，用**多个消费者从同一个主题读取消息，对消息进行分流。**

消费者与消费者组的关系：Kafka 的消费者都属于消费者组（consumer group）。一个组中的 consumer 订阅同样的 topic，每个 consumer 接收 topic 一些分区（partition）中的消息。**同一个分区不能被一个组中的多个 consumer 消费。**换句话说：**每一个分区内的消息，只能被同消费组中的一个消费者消费。**

Consumer Group

下图是一个topic有四个分区，随着消费者组中的消费者数量增加时，该topic分区中数据被读取的情况





运维层面

Leader: **分区**中的一个角色， Producer 和 Consumer 只与 Leader 交互；

Follower: **分区**中的一个角色，从 Leader 中复制数据，作为它的副本，同时一旦某 Leader 挂掉，便会从它的所有 Follower 中选举出一个新的 Leader 继续提供服务；

Zookeeper在Kafka中的作用: Kafka集群之间并不直接联系，而是将各自节点信息注册到 Zookeeper，由ZK进行统一管理。

小结: 多个broker组成一个Kafka集群，broker之间通过Zookeeper进行通讯，一个broker中可以有多个topic，生产者可以将消息发送至topic的不同分区中，同一个topic的不同分区内的消息不一样，每个分区可以有最多N（取决于broker的数量）个副本保存在不同的broker中。

消费者从订阅的topic中拉取消息，可以通过增加消费者形成消费者组来提高消费者的消费速度。

每个分区中有两个角色，类似主从，生产者和消费者只与leader节点交互，leader节点出故障后，会从所有的follower节点中选举出一个新的leader节点继续提供服务。



PART 03

搭建



环境

本次搭建环境

CentOS Linux release 7.3.1611 (Core)

java version "1.8.0_321"

kafka_2.12-2.7.0(使用了内置的zookeeper)

Kafka下载地址

<https://kafka.apache.org/downloads>

注意：Kafka从3.0.0，不再支持Java8



目录结构

| 目录名称 | 说明 |
|-----------|---------------------------|
| bin | kafka所有的执行脚本 |
| config | kafak所有的配置文件(包括zookeeper) |
| logs | kafka所有日志文件 |
| libs | kafka所依赖的所有jar包 |
| site-docs | 使用说明文档 |



新建项

创建日志文件夹

因为日志默认是存在/tmp 下，重新启动后会消失

kafka日志文件：/home/app/kafka/kafka_2.12-2.7.0/log/kafka

zookeeper日志文件：/home/app/kafka/kafka_2.12-2.7.0/log/zookeeper

创建zookeeper数据文件夹

/home/app/kafka/kafka_2.12-2.7.0/zkData

在zookeeper数据文件夹中创建文件

其中 1 则是每个服务器的id，不可重复

```
cd /home/app/kafka/kafka_2.12-2.7.0/zkData
```

```
echo 1 > myid
```



zookeeper.properties

修改config文件夹下zookeeper.properties配置文件

```
# 数据目录，对应新创建的数据目录
dataDir=/home/app/kafka/kafka_2.12-2.7.0/zkData
# 日志目录，对应新创建的目录
dataLogDir=/home/app/kafka/kafka_2.12-2.7.0/log/zookeeper
# 注释该行
# maxClientCnxns=0

# 设置连接参数，添加如下配置
# 为zk的基本时间单元，毫秒
tickTime=2000
# Leader-Follower初始通信时限 tickTime*10
initLimit=10
# Leader-Follower同步通信时限 tickTime*5
syncLimit=5
# 设置broker Id的服务地址（修改对应服务器ip即可）
# 其中server.0 server.1 server.2 后的 0 1 2 也是对应zookeeper数据目录zkData下myid文件中的内容
server.0=192.168.92.128:2888:3888
server.1=192.168.92.129:2888:3888
server.2=192.168.92.130:2888:3888
```



server.properties

修改config文件夹下server.properties Kafka的配置文件

```
# broker 的全局唯一编号，不能重复，对应 zkData中myid的值
broker.id=0
# 配置监听，修改位本机ip
listeners=PLAINTEXT://192.168.92.128:9092
advertised.listeners=PLAINTEXT://192.168.92.128:9092
# 修改输出日志文件位置
log.dirs=/home/app/kafka/kafka_2.12-2.7.0/log/kafka
# 配置三台zookeeper地址
zookeeper.connect=192.168.92.128:2181,192.168.92.129:2181,192.168.92.130:2181
```



启动

需要先启动zookeeper,等zookeeper启动完成后启动kafka

启动zookeeper

当前命令行启动

```
./bin/zookeeper-server-start.sh ./config/zookeeper.properties &
```

后台启动

```
nohup ./bin/zookeeper-server-start.sh ./config/zookeeper.properties > ./log/zookeeper/zookeeper.log 2>1 &
```

启动kafka

当前命令行启动

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

后台启动

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



PART 04

Java API使用





Java API使用

wiki地址:

<http://192.168.0.101:8090/pages/viewpage.action?pageId=24478527>



ACKS

这里的acks指的是**producer**的消息发送确认机制。

acks有三个值可以选择 0, 1, -1 (all)

acks = 0: 就是kafka生产端发送消息之后, 不管broker有没有成功收到消息, 在producer端都会认为是发送成功了, 这种情况提供了最小的延迟, 和最弱的持久性, 如果在发送途中leader异常, 就会造成数据丢失。

acks = 1: 是kafka默认的消息发送确认机制, 此机制是在producer发送数据成功, 并且leader接收成功并确认后就算消息发送成功, 但是这种情况如果leader接收成功了, 但是follower未同步时leader异常, 就会造成上位的follower丢失数据, 提供了较好的持久性和较低的延迟性。

acks = -1: 也可以设置成all, 此机制是producer发送成功数据, 并且leader接收成功, 并且follower也同步成功之后, producer才会发送下一条数据。



消费者消息确认

自动提交

最简单的方式是消费者自动提交偏移量。如果 **enable.auto.commit** 设为 **true**，那么每隔一定时间间隔，消费者会自动把从 `poll()` 方法接收到的最大偏移量提交上去。提交时间间隔由 **auto.commit.interval.ms** 控制，默认是5s。

缺点：假如在提交时间间隔内发生了分区再均衡（比如topic添加分区，会将消息进行重新分配），会发生消息重新被消费的情况，所以只能通过调小提交时间间隔来更频繁的提交偏移量，但也无法完全避免。



消费者消息确认

手动提交

同步提交

把 `auto.commit.offset` 设为 `false`，使用 `commitSync()` 方法提交偏移量最简单也最可靠，该方法会提交由 `poll()` 方法返回的最新偏移量，提交成功后马上返回，如果提交失败就抛出异常。需在 `poll()` 的所有数据处理完成后再调用。只要没有发生不可恢复的错误，`commitSync()` 会一直尝试直至提交成功。如果提交失败会抛出 `CommitFailedException` 异常。

异步提交

手动提交有一个不足之处，在 broker 对提交请求作出回应之前，应用程序会阻塞，这会影响应用程序的吞吐量。可以使用异步提交的方式，不等待 broker 的响应。

异步提交是不能重试的，因为重试的时候，待提交的位移很可能是一个过期的位移（也就是偏移量值大的有可能会比偏移量值小的先提交）。对于失败或者成功后续的处理，可以在定义的回调函数中进行处理。

方法： `consumer.commitAsync()`



Thank you!

