一 Kafka概述

1.1 Kafka是什么

在流式计算中，Kafka一般用来缓存数据，Storm通过消费Kafka的数据进行计算。

1）Apache Kafka是一个开源**消息**系统，由Scala写成。是由Apache软件基金会开发的一个开源消息系统项目。

2）Kafka最初是由LinkedIn公司开发，并于 2011年初开源。2012年10月从Apache Incubator毕业。该项目的目标是为处理实时数据提供一个统一、高通量、低等待的平台。

3）**Kafka是一个分布式消息队列。**Kafka对消息保存时根据Topic进行归类，发送消息者称为Producer，消息接受者称为Consumer，此外kafka集群有多个kafka实例组成，每个实例(server)成为broker。

4）无论是kafka集群，还是producer和consumer都依赖于**zookeeper**集群保存一些meta信息，来保证系统可用性。

kafka认识：

是一个消息系统

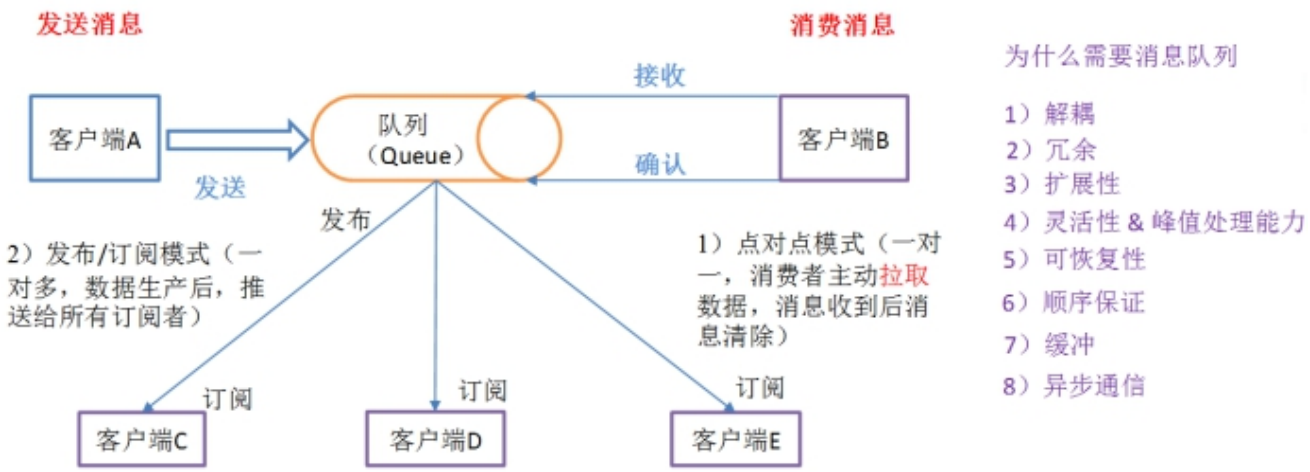
它有存储的能力

对收到的每一条消息有处理能力

使用发布订阅模式，消费者消费数据使用拉模式

生产者发送数据的时候，指定topic（类别），消费的指定要消费的topic。

1.2 消息队列内部实现原理



（1）点对点模式（一对一，消费者主动拉取数据，消息收到后消息清除）

点对点模型通常是一个基于拉取或者轮询的消息传送模型，这种模型从队列中请求信息，而不是将消息推送到客户端。这个模型的特点是发送到队列的消息被一个且只有一个接收者接收处理，即使有多个消息监听者也是如此。

（2）发布/订阅模式（一对多，数据生产后，推送给所有订阅者）

发布订阅模型则是一个基于推送的消息传送模型。发布订阅模型可以有多种不同的订阅者，临时订阅者只在主动监听主题时才接收消息，而持久订阅者则监听主题的所有消息，即使当前订阅者不可用，处于离线状态。

1.3 为什么需要消息队列

1）解耦：

允许你独立的扩展或修改两边的处理过程，只要确保它们遵守同样的接口约束。

2）冗余：

消息队列把数据进行持久化直到它们已经被完全处理，通过这一方式规避了数据丢失风险。许多消息队列所采用的"插入-获取-删除"范式中，在把一个消息从队列中删除之前，需要你的处理系统明确的指出该消息已经被处理完毕，从而确保你的数据被安全的保存直到你使用完毕。

3）扩展性：

因为消息队列解耦了你的处理过程，所以增大消息入队和处理的频率是很容易的，只要另外增加处理过程即可。

4）灵活性 & 峰值处理能力：

在访问量剧增的情况下，应用仍然需要继续发挥作用，但是这样的突发流量并不常见。如果为以能处理这类峰值访问为标准来投入资源随时待命无疑是巨大的浪费。使用消息队列能够使关键组件顶住突发的访问压力，而不会因为突发的超负荷的请求而完全崩溃。

5）可恢复性：

系统的一部分组件失效时，不会影响到整个系统。消息队列降低了进程间的耦合度，所以即使一个处理消息的进程挂掉，加入队列中的消息仍然可以在系统恢复后被处理。

6）顺序保证：

在大多使用场景下，数据处理的顺序都很重要。大部分消息队列本来就是排序的，并且能保证数据会按照特定的顺序来处理。（Kafka保证一个Partition内的消息的有序性）

7）缓冲：

有助于控制和优化数据流经过系统的速度，解决生产消息和消费消息的处理速度不一致的情况。

8）异步通信：

很多时候，用户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。

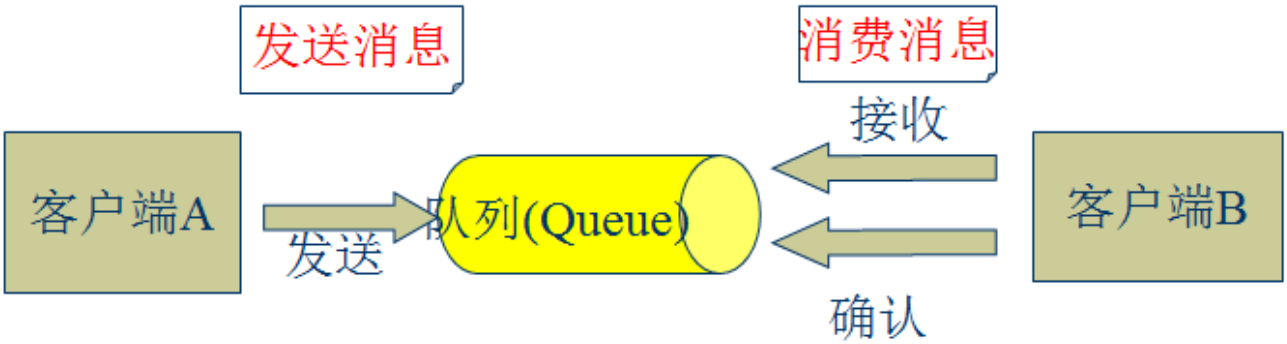
1.4 JMS规范介绍

JMS概念

JMS：是Java提供的一套技术规范。

JMS用途：用来异构系统集成通信，缓解系统瓶颈，提高系统的伸缩性增强系统用户体验，使得系统模块化和组件化变得可行并更加灵活。

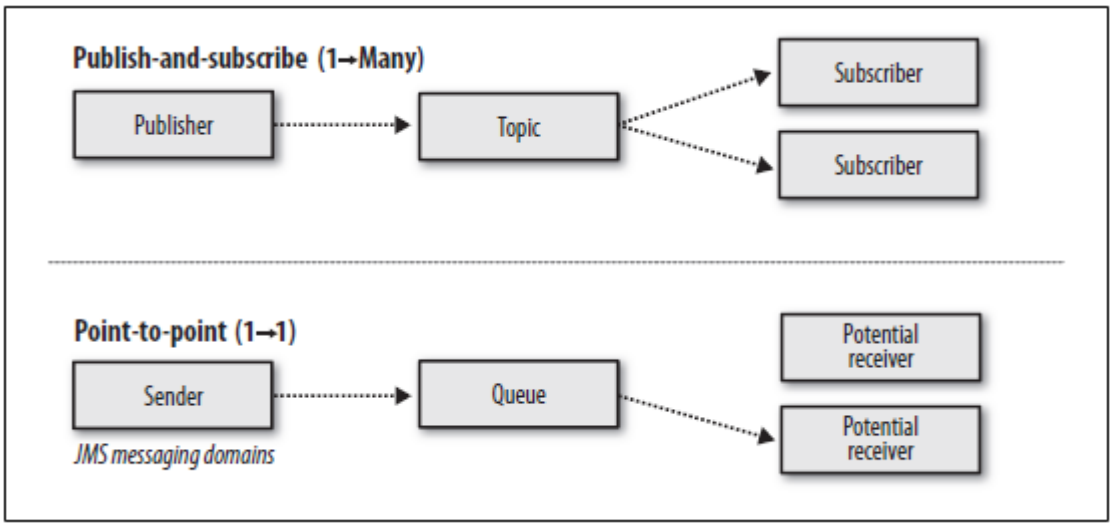
实现方式：生产消费者模式（生产者、服务器、消费者）



JMS消息传输模型

点对点模式（一对一，消费者主动拉取数据，消息收到后消息清除）
点对点模型通常是一个基于拉取或者轮询的消息传送模型，这种模型从队列中请求信息，而不是将消息推送到客户端。这个模型的特点是发送到队列的消息被**一个且只有一个接收者接收处理，即使有多个消息监听者也是如此。

发布/订阅模式（一对多，数据生产后，推送给所有订阅者）
发布订阅模型则是一个基于推送的消息传送模型。发布订阅模型可以有多种不同的订阅者，临时订阅者只在主动监听主题时才接收消息，而持久订阅者则监听主题的所有消息，即当前订阅者不可用，处于离线状态。

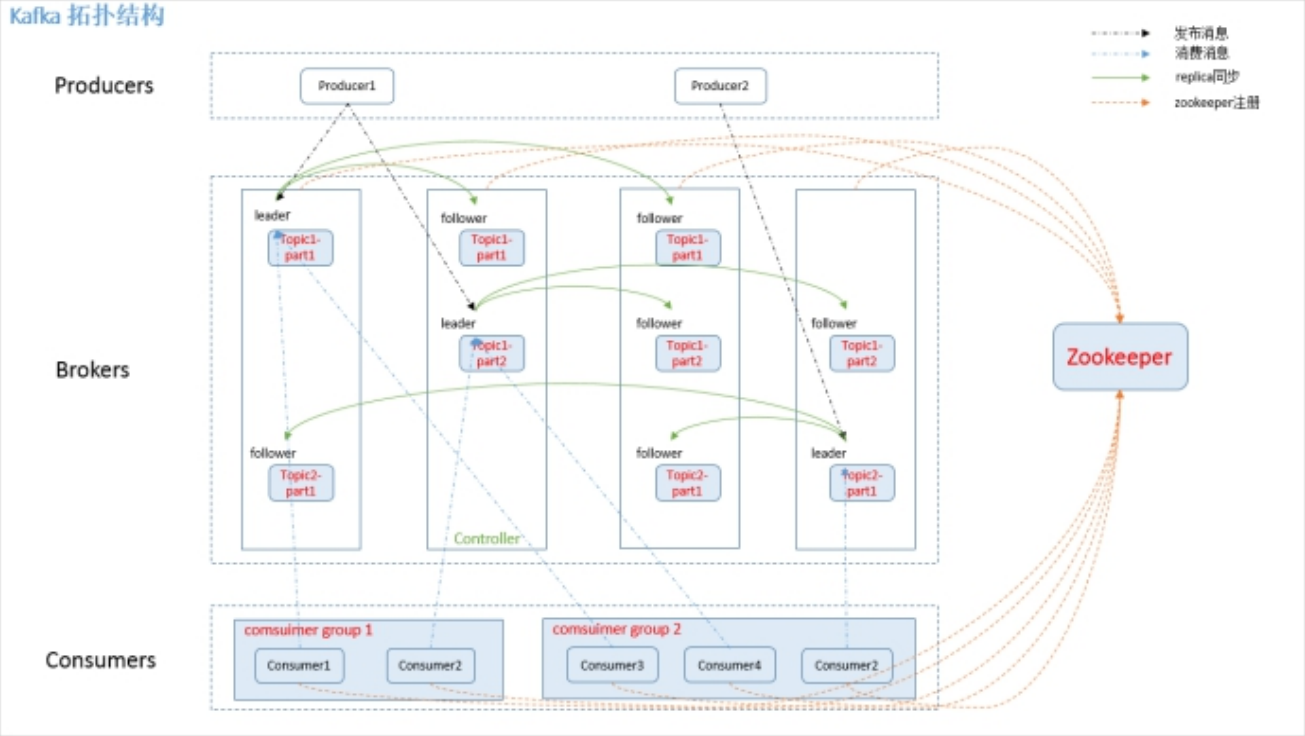


JMS核心组件

`Destination`：消息发送的目的地，也就是前面说的`Queue`和`Topic`。
`Message` ：从字面上就可以看出是被发送的消息。
`Producer`： 消息的生产者，要发送一个消息，必须通过这个生产者来发送。
`MessageConsumer`： 与生产者相对应，这是消息的消费者或接收者，通过它来接收一个消息。

StreamMessage：Java 数据流消息，用标准流操作来顺序的填充和读取。
MapMessage：一个Map类型的消息，名称为 string 类型，而值为 Java 的基本类型。
TextMessage：普通字符串消息，包含一个String。
ObjectMessage：对象消息，包含一个可序列化的Java对象
BytesMessage：二进制数组消息，包含一个byte[]。
XMLMessage: 一个XML类型的消息。
最常用的是TextMessage和ObjectMessage。

1.5 Kafka架构



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）Topic ：可以理解为一个队列。

4） Consumer Group （CG）：这是kafka用来实现一个topic消息的广播（发给所有的consumer）和单播（发给任意一个consumer）的手段。一个topic可以有多个CG。topic的消息会复制-给consumer。如果需要实现广播，只要每个consumer有一个独立的CG就可以了。要实现单播只要所有的consumer在同一个CG。用CG还可以将consumer进行自由的分组而不需要多次发送消息到不同的topic。

5）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。

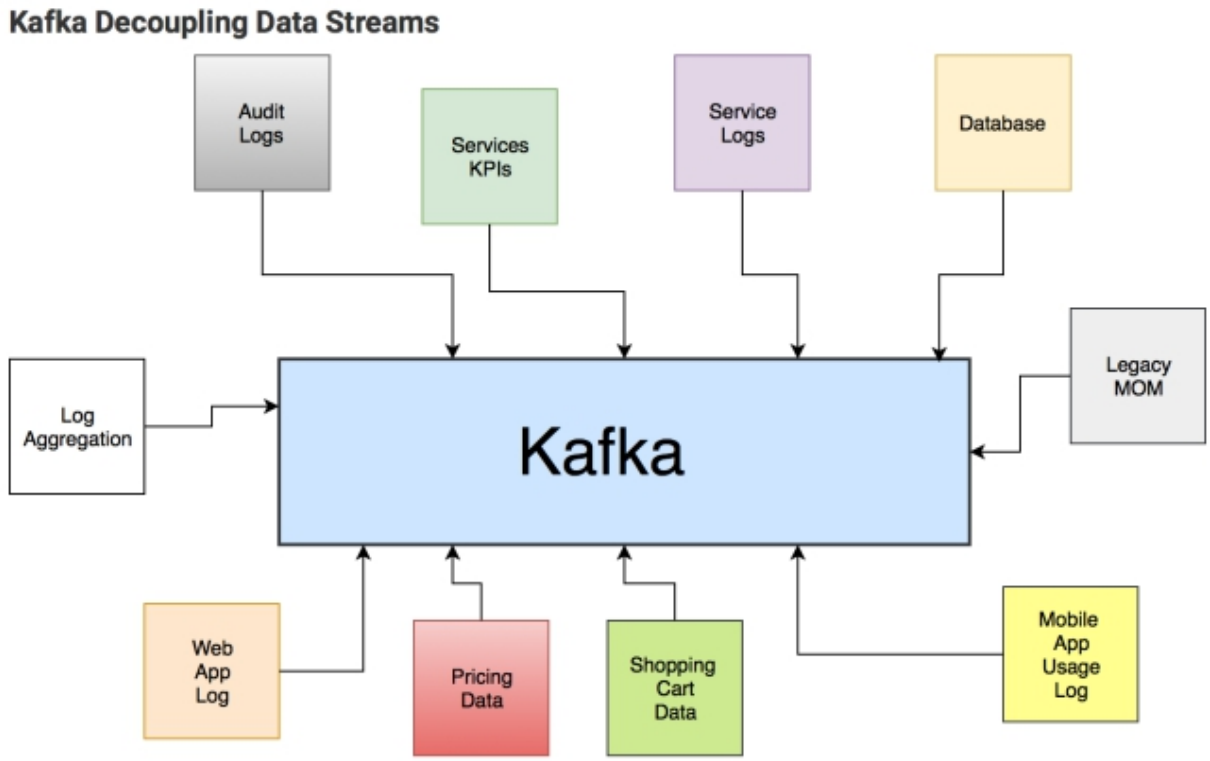
6）Partition：为了实现扩展性，一个非常大的topic可以分布到多个broker（即服务器）上，一个topic可以分为多个partition，每个partition是一个有序的队列。partition中的每条消息都会被分配一个有序的id（offset）。kafka只保证按一个partition中的顺序将消息发给consumer，不保证一个topic的整体（多个partition间）的顺序。

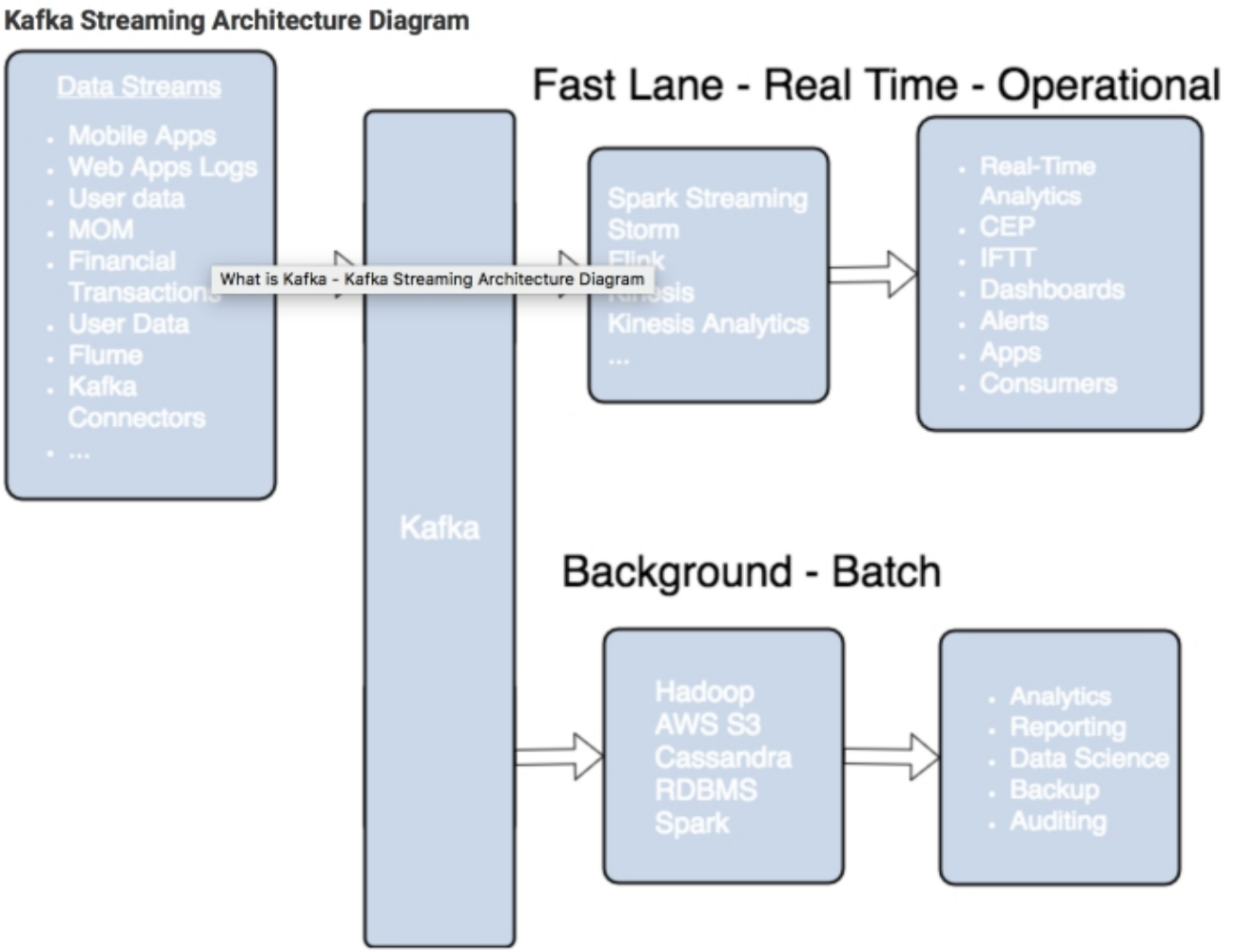
7）Offset：kafka的存储文件都是按照offset.kafka来命名，用offset做名字的好处是方便查找。例如你想找位于2049的位置，只要找到2048.kafka的文件即可。当然the first offset就是00000000000.kafka

一个典型的kafka集群中包含若干producer（可以是web前端产生的page view，或者是服务器日志，系统CPU、memory等），若干broker（Kafka支持水平扩展，一般broker数量越多，集群吞吐率越高），若干consumer group，以及一个Zookeeper集群。Kafka通过Zookeeper管理集群配置，选举leader，以及在consumer group发生变化时进行rebalance。producer使用push模式将消息发布到broker，consumer使用pull模式从broker订阅并消费消息。

1.6 kafka流架构

Kafka 经常在实时流数据处理系统中，在实时流数据处理系统中，kafka作为一个中间层解耦实时数据流管道。





1.7 分布式模型

Kafka每个主题的多个分区日志分布式地存储在Kafka集群上，同时为了故障容错，每个分区都会以副本的方式复制到多个消息代理节点上。其中一个节点会作为主副本（Leader），其他节点作为备份副本（Follower，也叫作从副本）。主副本会负责所有的客户端读写操作，备份副本仅仅从主副本同步数据。当主副本出现故障时，备份副本中的一个副本会被选择为新的主副本。因为每个分区的副本中只有主副本接受读写，所以每个服务器端都会作为某些分区的主副本，以及另外一些分区的备份副本，这样Kafka集群的所有服务端整体上对客户端是负载均衡的。

Kafka的生产者和消费者相对于服务器端而言都是客户端。

Kafka生产者客户端发布消息到服务端的指定主题，会指定消息所属的分区。生产者发布消息时根据消息是否有键，采用不同的分区策略。消息没有键时，通过轮询方式进行客户端负载均衡；消息有键时，根据分区语义（例如hash）确保相同键的消息总是发送到同一分区。

Kafka的消费者通过订阅主题来消费消息，并且每个消费者都会设置一个消费组名称。因为生产者发布到主题的每一条消息都只会发送给消费者组的一个消费者。所以，如果要实现传统消息系统的“队列”模型，可以让每个消费者都拥有相同的消费组名称，这样消息就会负责均衡到所有的消费者；如果要实现“发布-订阅”模型，则每个消费者的消费者组名称都不相同，这样每条消息就会广播给所有的消费者。

分区是消费者现场模型的最小并行单位。如下图（图1）所示，生产者发布消息到一台服务器的3个分区时，只有一个消费者消费所有的3个分区。在下图（图2）中，3个分区分布在3台服务器上，同时有3个消费者分别消费不同的分区。假设每个服务器的吞吐量时300MB，在下图（图1）中分摊到每个分区只有100MB，而在下图（图2）中，集群整体的吞吐量有900MB。可以看到，增加服务器节点会提升集群的性能，增加消费者数量会提升处理性

能。

同一个消费组下多个消费者互相协调消费工作，Kafka会将所有的分区平均地分配给所有的消费者实例，这样每个消费者都可以分配到数量均等的分区。Kafka的消费组管理协议会动态地维护消费组的成员列表，当一个新消费者加入消费者组，或者有消费者离开消费组，都会触发再平衡操作。

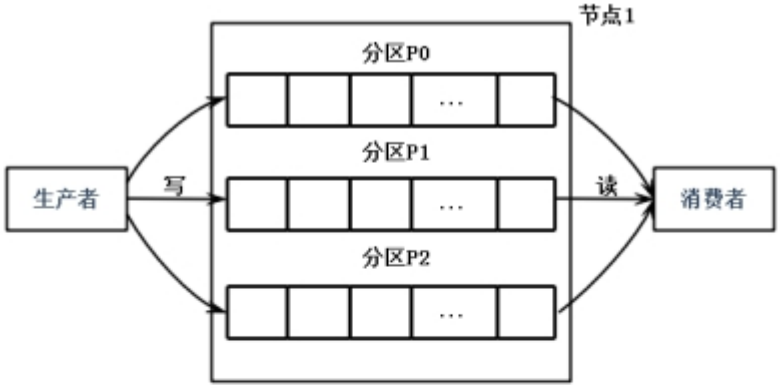


图1

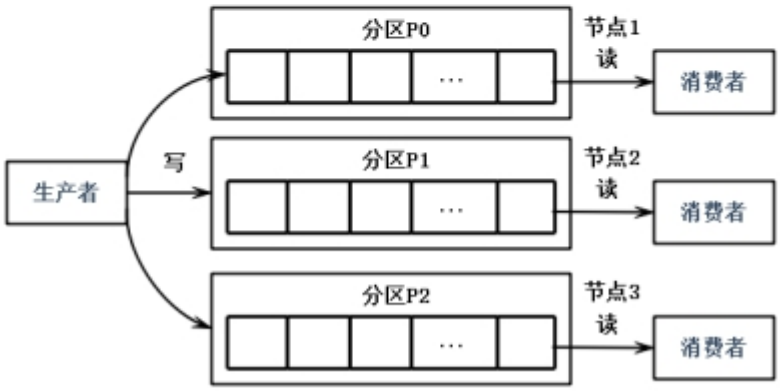


图2

Kafka的消费者消费消息时，只保证在一个分区内的消息的完全有序性，并不保证同一个主题汇中多个分区的消息顺序。而且，消费者读取一个分区消息的顺序和生产者写入到这个分区的顺序是一致的。比如，生产者写入“hello”和“Kafka”两条消息到分区P1，则消费者读取到的顺序也一定是“hello”和“Kafka”。如果业务上需要保证所有消息完全一致，只能通过设置一个分区完成，但这种做法的缺点是最多只能有一个消费者进行消费。一般来说，只需要保证每个分区的有序性，再对消息假设键来保证相同键的所有消息落入同一分区，就可以满足绝大多数的应用。

1.8 Kafka核心组件

producer

消息生产者，负责将数据写入（push）到broker，或者说将消息发布到 kafka 集群。

指定消息的类别（topic）,和分区里的leader交互（写消息）

consumer

消息的消费者，负责从kafka读取（pull）数据，老版本（0.9.0.0版本之前），消费者依赖zookeeper保存一

些信息，如消费者群组的信息、主题信息、消费分区的偏移量；,新版本(0.9.0.0以及之后)引入了新的消费者接

口，允许broker直接维护这些信息，不再依赖zookeeper存储这些信息。

Consumer group

每个consumer属于一个特定的consumer group（可为每个consumer指定group name，若不指定group name则属于默认的group）。

消费者组去订阅某个topic

一个消费者组里的消费者怎么消费数据？

一条topic可以被多个消费者同时消费，但是若多个消费者同属于一个 consumer group，每条消息只能被 consumer group 中的一个 Consumer 消费，但可以被多个 consumer group 消费。

（一条消息可以被多个消费者消费，每个消费者也可以消费多条消息，一条消息只能被消费者组里的某一个消费者消费，这个组其他消费者不能再消费同一条消息，但是不同消费者组里的消费者可以多次消费同一条消息）

Broker 安装kafka服务的那台机器就叫一个broker，Kafka集群包含一个或多个服务器（broker），每个broker的id 在集群中全局唯一，每个broker可以容纳多个topic。

Topic 相当于数据的一个分类，不同的topic存放不同的数据，每条发布到Kafka集群的消息都有一个类别，这个类别被称为topic。（物理上不同topic的消息分开存储，逻辑上一个topic的消息虽然保存于一个或多个broker上但用户只需指定消息的topic即可生产或消费数据而不必关心数据存于何处）

Partition

parition是物理上的概念，每个topic包含一个或多个partition，创建topic时可指定parition数量。每个partition对应于一个文件夹，该文件夹下存储该partition的数据和索引文件

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。partition中的每条消息都会被分配一个有序的id（offffset）。

kafka只保证按一个partition中的顺序将消息发给consumer，不保证一个topic的整体（多个partition间）的顺序。

replica：partition 的副本，保障 partition 的高可用。

leader：replica 中的一个角色， producer 和 consumer 只跟 leader 交互。

follower：replica 中的一个角色，从 leader 中复制数据。

Offset：kafka的存储文件都是按照offset.kafka来命名，用offset做名字的好处是方便查找。例如你想找位于2049的位置，只要找到2048.kafka的文件即可。当然the fifirst offset就是00000000000.kafka。

二 Kafka集群部署

2.1 环境准备

2.1.1 集群规划

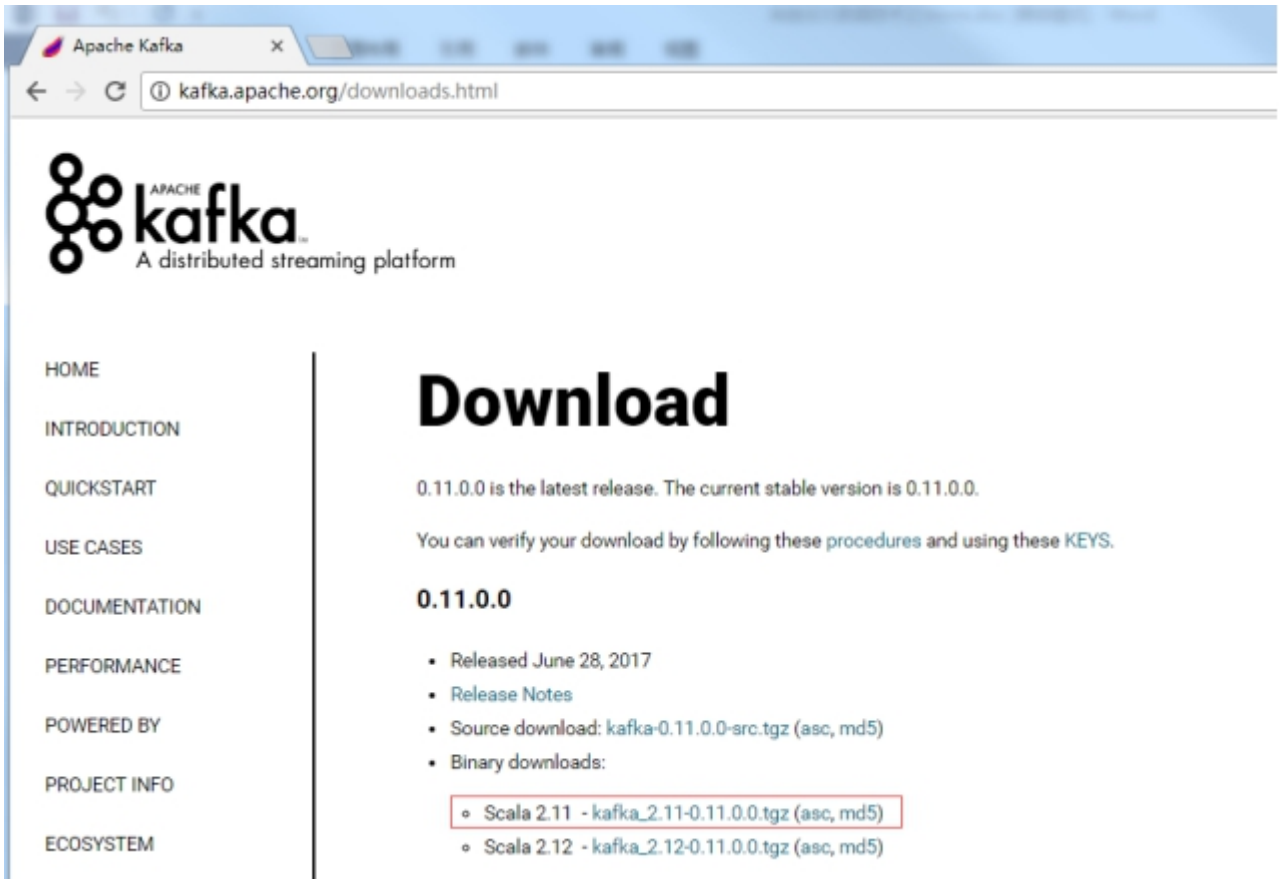
hadoop102 hadoop103 hadoop104

zk zk zk

kafka kafka kafka

2.1.2 jar包下载

http://kafka.apache.org/downloads.html



2.1.3 虚拟机准备

1）准备3台虚拟机

2）配置ip地址

3）配置主机名称

4）3台主机分别关闭防火墙

[root@hadoop102]# chkconfig iptables off

[root@hadoop103]# chkconfig iptables off

[root@hadoop104]# chkconfig iptables off

2.1.4 安装jdk

2.1.5 安装Zookeeper

0）集群规划

在hadoop102、hadoop103和hadoop104三个节点上部署Zookeeper。

1）解压安装

（1）解压zookeeper安装包到/opt/module/目录下

[root@hadoop102 software]$ tar -zxvf zookeeper-3.4.10.tar.gz -C /opt/module/

（2）在/opt/module/zookeeper-3.4.10/这个目录下创建zkData

mkdir -p zkData

（3）重命名/opt/module/zookeeper-3.4.10/conf这个目录下的zoo_sample.cfg为zoo.cfg

mv zoo_sample.cfg zoo.cfg

2）配置zoo.cfg文件

（1）具体配置

dataDir=/opt/module/zookeeper-3.4.10/zkData

增加如下配置

#######################cluster##########################

server.2=hadoop102:2888:3888

server.3=hadoop103:2888:3888

server.4=hadoop104:2888:3888

（2）配置参数解读

Server.A=B:C:D。

A是一个数字，表示这个是第几号服务器；

B是这个服务器的ip地址；

C是这个服务器与集群中的Leader服务器交换信息的端口；

D是万一集群中的Leader服务器挂了，需要一个端口来重新进行选举，选出一个新的Leader，而这个端口就是用来执行选举时服务器相互通信的端口。

集群模式下配置一个文件myid，这个文件在dataDir目录下，这个文件里面有一个数据就是A的值，Zookeeper启动时读取此文件，拿到里面的数据与zoo.cfg里面的配置信息比较从而判断到底是哪个server。

3）集群操作

（1）在/opt/module/zookeeper-3.4.10/zkData目录下创建一个myid的文件

touch myid

添加myid文件，注意一定要在linux里面创建，在notepad++里面很可能乱码

（2）编辑myid文件

vi myid

在文件中添加与server对应的编号：如2

（3）拷贝配置好的zookeeper到其他机器上

scp -r zookeeper-3.4.10/...

scp -r zookeeper-3.4.10/ ...

并分别修改myid文件中内容为3、4

（4）分别启动zookeeper

[root@hadoop102 zookeeper-3.4.10]# bin/zkServer.sh start

[root@hadoop103 zookeeper-3.4.10]# bin/zkServer.sh start

[root@hadoop104 zookeeper-3.4.10]# bin/zkServer.sh start

（5）查看状态

[root@hadoop102 zookeeper-3.4.10]# bin/zkServer.sh status

JMX enabled by default

Using config: /opt/module/zookeeper-3.4.10/bin/../conf/zoo.cfg

Mode: follower

[root@hadoop103 zookeeper-3.4.10]# bin/zkServer.sh status

JMX enabled by default

Using config: /opt/module/zookeeper-3.4.10/bin/../conf/zoo.cfg

Mode: leader

[root@hadoop104 zookeeper-3.4.5]# bin/zkServer.sh status

JMX enabled by default

Using config: /opt/module/zookeeper-3.4.10/bin/../conf/zoo.cfg

Mode: follower

2.2 Kafka集群部署

kafka集群安装前，需要预安装zookeeper。

1. 下载Kafka安装包
2. 解压 tar -xvf kafka_2.11-0.9.0.1.tar
3. 修改配置文件 config/server.properties

broker.id=0

host.name=centos0

log.dirs=/data/kafka

表示我们收集到的数据以log的形式存储在这个目录当中，这个一定要改，因为它默认的话给我们放在tmp

里，那每次启动后，数据就没有了。

zookeeper.connect=node-1:2181,node-2:2181,node-3:2181

4. 将配置好的kafka拷贝到其他机器上

5. 修改broker.id和host.name
6. 启动kafka:

/bigdata/kafka_2.11-0.9.0.1/bin/kafka-server-start.sh -daemon /bigdata/kafka_2.11-

0.9.0.1/config/server.properties

2.3 Kafka命令行操作

1）查看当前服务器中的所有topic

bin/kafka-topics.sh --zookeeper node1:2181 --list

2）创建topic

bin/kafka-topics.sh --zookeeper node1:2181 --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

3）删除topic

bin/kafka-topics.sh --zookeeper node1:2181 --delete --topic test

需要server.properties中设置delete.topic.enable=true否则只是标记删除或者直接重启。

4）发送消息

bin/kafka-console-producer.sh --broker-list hadoop102:9092 --topic first

>hello world

5）消费消息

bin/kafka-console-consumer.sh --zookeeper node1:2181 --from-beginning --topic test

--from-beginning：会把first主题中以往所有的数据都读取出来。根据业务场景选择是否增加该配置。

6）查看某个Topic的详情

bin/kafka-topics.sh --zookeeper hadoop102:2181 --describe --topic first

第一行显示partitions的概况，列出了Topic名字，partition总数，存储这些partition的broker数

以下每一行都是其中一个partition的详细信息：

leader

是该partitons所在的所有broker中担任leader的broker id，每个broker都有可能成为leader

bin/kafka-topics.sh --list --zookeeper node01:2181

bin/kafka-topics.sh --create --zookeeper node01:2181 --replication-factor 1 --partitions 1 --

topic test

bin/kafka-topics.sh --delete --zookeeper node01:2181 --topic test

bin/kafka-console-producer.sh --broker-list node01:9092 --topic test1

bin/kafka-console-consumer.sh --zookeeper node01:2181 --from-beginning --topic test1

bin/kafka-run-class.sh kafka.tools.ConsumerOffsetChecker --zookeeper node01:2181 --group

testGroup

bin/kafka-topics.sh --topic test --describe --zookeeper node01:2181

replicas

显示该partiton所有副本所在的broker列表，包括leader，不管该broker是否是存活，不管是否和leader保持了同步。

isr

in-sync replicas的简写，表示存活且副本都已同步的的broker集合，是replicas的子集

举例：

比如上面结果的第一行：Topic: test0 Partition:0 Leader: 0 Replicas: 0,2,1 Isr: 1,0,2

Partition: 0

该partition编号是0

Replicas: 0,2,1

代表partition0 在broker0，broker1，broker2上保存了副本

Isr: 1,0,2

代表broker0，broker1，broker2都存活而且目前都和leader保持同步

Leader: 0

代表保存在broker0，broker1，broker2上的这三个副本中，leader是broker0

leader负责读写，broker1、broker2负责从broker0同步信息，平时没他俩什么事

当producer发送一个消息时，producer自己会判断发送到哪个partiton上，如果发到了partition0上，消息会发到

leader，也就是broker0上，broker0处理这个消息，broker1、broker2从broker0同步这个消息

如果这个broker0挂了，那么kafka会在Isr列表里剩下的broker1、broker2中选一个新的leader

测试Kafka集群一共三个节点，test这个Topic, 编号为0的Partition,Leader在broker.id=0这个节点上，副本在

broker.id为0 1 2这个三个几点，并且所有副本都存活，并跟broker.id=0这个节点同步

对分区数进行修改（只能增加不能减少）

7) bin/kafka-topics.sh --zookeeper node01 --alter --partitions 15 --topic utopic

2.4 Kafka配置信息

2.4.1 Broker配置信息

属性	默认值	描述
broker.id		必填参数，broker的唯一标识
log.dirs	/tmp/kafka-logs	Kafka数据存放的目录。可以指定多个目录，中间用逗号分隔，当新partition被创建的时会被存放到当前存放partition最少的目录。
port	9092	BrokerServer接受客户端连接的端口号
zookeeper.connect	null	Zookeeper的连接串，格式为：hostname1:port1,hostname2:port2,hostname3:port3。可以填一个或多个，为了提高可靠性，建议都填上。注意，此配置允许我们指定一个zookeeper路径来存放此kafka集群的所有数据，为了与其他应用集群区分开，建议在此配置中指定本集群存放目录，格式为：hostname1:port1,hostname2:port2,hostname3:port3/chroot/path。需要注意的是，消费者的参数要和此参数一致。
message.max.bytes	1000000	服务器可以接收到的最大的消息大小。注意此参数要和consumer的maximum.message.size大小一致，否则会因为生产者生产的消息太大导致消费者无法消费。
num.io.threads	8	服务器用来执行读写请求的IO线程数，此参数的数量至少要等于服务器上磁盘的数量。
queued.max.requests	500	I/O线程可以处理请求的队列大小，若实际请求数超过此大小，网络线程将停止接收新的请求。
socket.send.buffer.bytes	100 * 1024	The SO_SNDBUFF buffer the server prefers for socket connections.
socket.receive.buffer.bytes	100 * 1024	The SO_RCVBUFF buffer the server prefers for socket connections.
socket.request.max.bytes	100 * 1024 * 1024	服务器允许请求的最大值， 用来防止内存溢出，其值应该小于 Java heap size.
num.partitions	1	默认partition数量，如果topic在创建时没有指定partition数量，默认使用此值，建议改为5
log.segment.bytes	1024 * 1024 * 1024	Segment文件的大小，超过此值将会自动新建一个segment，此值可以被topic级别的参数覆盖。
log.roll.{ms,hours}	24 * 7 hours	新建segment文件的时间，此值可以被topic级别的参数覆盖。
log.retention.{ms,minutes,hours}	7 days	Kafka segment log的保存周期，保存周期超过此时间日志就会被删除。此参数可以被topic级别参数覆盖。数据量大时，建议减小此值。
log.retention.bytes	-1	每个partition的最大容量，若数据量超过此值，partition数据将会被删除。注意这个参数控制的是每个partition而不是topic。此参数可以被log级别参数覆盖。
log.retention.check.interval.ms	5 minutes	删除策略的检查周期
auto.create.topics.enable	true	自动创建topic参数，建议此值设置为false，严格控制topic管理，防止生产者错写topic。
default.replication.factor	1	默认副本数量，建议改为2。
replica.lag.time.max.ms	10000	在此窗口时间内没有收到follower的fetch请求，leader会将其从ISR(in-sync replicas)中移除。
replica.lag.max.messages	4000	如果replica节点落后leader节点此值大小的消息数量，leader节点就会将其从ISR中移除。
replica.socket.timeout.ms	30 * 1000	replica向leader发送请求的超时时间。
replica.socket.receive.buffer.bytes	64 * 1024	The socket receive buffer for network requests to the leader for replicating data.
replica.fetch.max.bytes	1024 * 1024	The number of byes of messages to attempt to fetch for each partition in the fetch requests the replicas send to the leader.
replica.fetch.wait.max.ms	500	The maximum amount of time to wait time for data to arrive on the leader in the fetch requests sent by the replicas to the leader.
num.replica.fetchers	1	Number of threads used to replicate messages from leaders. Increasing this value can increase the degree of I/O parallelism in the follower broker.

属性	默认值	描述
fetch.purgatory.purge.interval.requests	1000	The purge interval (in number of requests) of the fetch request purgatory.
zookeeper.session.timeout.ms	6000	ZooKeeper session 超时时间。如果在此时间内server没有向zookeeper发送心跳，zookeeper就会认为此节点已挂掉。 此值太低导致节点容易被标记死亡；若太高，.会导致太迟发现节点死亡。
zookeeper.connection.timeout.ms	6000	客户端连接zookeeper的超时时间。
zookeeper.sync.time.ms	2000	H ZK follower落后 ZK leader的时间。
controlled.shutdown.enable	true	允许broker shutdown。如果启用，broker在关闭自己之前会把它上面的所有leaders转移到其它brokers上，建议启用，增加集群稳定性。
auto.leader.rebalance.enable	true	If this is enabled the controller will automatically try to balance leadership for partitions among the brokers by periodically returning leadership to the “preferred” replica for each partition if it is available.
leader.imbalance.per.broker.percentage	10	The percentage of leader imbalance allowed per broker. The controller will rebalance leadership if this ratio goes above the configured value per broker.
leader.imbalance.check.interval.seconds	300	The frequency with which to check for leader imbalance.
offset.metadata.max.bytes	4096	The maximum amount of metadata to allow clients to save with their offsets.
connections.max.idle.ms	600000	Idle connections timeout: the server socket processor threads close the connections that idle more than this.
num.recovery.threads.per.data.dir	1	The number of threads per data directory to be used for log recovery at startup and flushing at shutdown.
unclean.leader.election.enable	true	Indicates whether to enable replicas not in the ISR set to be elected as leader as a last resort, even though doing so may result in data loss.
delete.topic.enable	false	启用deletetopic参数，建议设置为true。
offsets.topic.num.partitions	50	The number of partitions for the offset commit topic. Since changing this after deployment is currently unsupported, we recommend using a higher setting for production (e.g., 100-200).
offsets.topic.retention.minutes	1440	Offsets that are older than this age will be marked for deletion. The actual purge will occur when the log cleaner compacts the offsets topic.
offsets.retention.check.interval.ms	600000	The frequency at which the offset manager checks for stale offsets.
offsets.topic.replication.factor	3	The replication factor for the offset commit topic. A higher setting (e.g., three or four) is recommended in order to ensure higher availability. If the offsets topic is created when fewer brokers than the replication factor then the offsets topic will be created with fewer replicas.
offsets.topic.segment.bytes	104857600	Segment size for the offsets topic. Since it uses a compacted topic, this should be kept relatively low in order to facilitate faster log compaction and loads.
offsets.load.buffer.size	5242880	An offset load occurs when a broker becomes the offset manager for a set of consumer groups (i.e., when it becomes a leader for an offsets topic partition). This setting corresponds to the batch size (in bytes) to use when reading from the offsets segments when loading offsets into the offset manager’s cache.
offsets.commit.required.acks	-1	The number of acknowledgements that are required before the offset commit can be accepted. This is similar to the producer’s acknowledgement setting. In general, the default should not be overridden.
offsets.commit.timeout.ms	5000	The offset commit will be delayed until this timeout or the required number of replicas have received the offset commit. This is similar to the producer request timeout.

2.4.2 Producer配置信息

属性	默认值	描述
metadata.broker.list		启动时producer查询brokers的列表，可以是集群中所有brokers的一个子集。注意，这个参数只是用来获取topic的元信息用，producer会从元信息中挑选合适的broker并与之建立socket连接。格式是：host1:port1,host2:port2。
request.required.acks	0	参见3.2节介绍
request.timeout.ms	10000	Broker等待ack的超时时间，若等待时间超过此值，会返回客户端错误信息。
producer.type	sync	同步异步模式。async表示异步，sync表示同步。如果设置成异步模式，可以允许生产者以batch的形式push数据，这样会极大的提高broker性能，推荐设置为异步。
serializer.class	kafka.serializer.DefaultEncoder	序列号类，.默认序列化成 byte[] 。
key.serializer.class		Key的序列化类，默认同上。
partitioner.class	kafka.producer.DefaultPartitioner	Partition类，默认对key进行hash。
compression.codec	none	指定producer消息的压缩格式，可选参数为： “none”, “gzip” and “snappy”。关于压缩参见4.1节
compressed.topics	null	启用压缩的topic名称。若上面参数选择了一个压缩格式，那么压缩仅对本参数指定的topic有效，若本参数为空，则对所有topic有效。
message.send.max.retries	3	Producer发送失败时重试次数。若网络出现问题，可能会导致不断重试。
retry.backoff.ms	100	Before each retry, the producer refreshes the metadata of relevant topics to see if a new leader has been elected. Since leader election takes a bit of time, this property specifies the amount of time that the producer waits before refreshing the metadata.

属性	默认值	描述
topic.metadata.refresh.interval.ms	600 * 1000	The producer generally refreshes the topic metadata from brokers when there is a failure (partition missing, leader not available...). It will also poll regularly (default: every 10min so 600000ms). If you set this to a negative value, metadata will only get refreshed on failure. If you set this to zero, the metadata will get refreshed after each message sent (not recommended). Important note: the refresh happen only AFTER the message is sent, so if the producer never sends a message the metadata is never refreshed
queue.buffering.max.ms	5000	启用异步模式时，producer缓存消息的时间。比如我们设置成1000时，它会缓存1秒的数据再一次发送出去，这样可以极大的增加broker吞吐量，但也会造成时效性的降低。
queue.buffering.max.messages	10000	采用异步模式时producer buffer 队列里最大缓存的消息数量，如果超过这个数值，producer就会阻塞或者丢掉消息。
queue.enqueue.timeout.ms	-1	当达到上面参数值时producer阻塞等待的时间。如果值设置为0，buffer队列满时producer不会阻塞，消息直接被丢掉。若值设置为-1，producer会被阻塞，不会丢消息。
batch.num.messages	200	采用异步模式时，一个batch缓存的消息数量。达到这个数量值时producer才会发送消息。
send.buffer.bytes	100 * 1024	Socket write buffer size
client.id	“”	The client id is a user-specified string sent in each request to help trace calls. It should logically identify the application making the request.

2.4.3 Consumer配置信息

属性	默认值	描述
group.id		Consumer的组ID，相同goup.id的consumer属于同一个组。
zookeeper.connect		Consumer的zookeeper连接串，要和broker的配置一致。
consumer.id	null	如果不设置会自动生成。
socket.timeout.ms	30 * 1000	网络请求的socket超时时间。实际超时时间由max.fetch.wait + socket.timeout.ms 确定。
socket.receive.buffer.bytes	64 * 1024	The socket receive buffer for network requests.
fetch.message.max.bytes	1024 * 1024	查询topic-partition时允许的最大消息大小。consumer会为每个partition缓存此大小的消息到内存，因此，这个参数可以控制consumer的内存使用量。这个值应该至少比server允许的最大消息大小大，以免producer发送的消息大于consumer允许的消息。
num.consumer.fetchers	1	The number fetcher threads used to fetch data.
auto.commit.enable	true	如果此值设置为true，consumer会周期性的把当前消费的offset值保存到zookeeper。当consumer失败重启之后将会使用此值作为新开始消费的值。
auto.commit.interval.ms	60 * 1000	Consumer提交offset值到zookeeper的周期。
queued.max.message.chunks	2	用来被consumer消费的message chunks 数量， 每个chunk可以缓存fetch.message.max.bytes大小的数据量。
auto.commit.interval.ms	60 * 1000	Consumer提交offset值到zookeeper的周期。
queued.max.message.chunks	2	用来被consumer消费的message chunks 数量， 每个chunk可以缓存fetch.message.max.bytes大小的数据量。
fetch.min.bytes	1	The minimum amount of data the server should return for a fetch request. If insufficient data is available the request will wait for that much data to accumulate before answering the request.
fetch.wait.max.ms	100	The maximum amount of time the server will block before answering the fetch request if there isn't sufficient data to immediately satisfy fetch.min.bytes.

属性	默认值	描述
rebalance.backoff.ms	2000	Backoff time between retries during rebalance.
refresh.leader.backoff.ms	200	Backoff time to wait before trying to determine the leader of a partition that has just lost its leader.
auto.offset.reset	largest	What to do when there is no initial offset in ZooKeeper or if an offset is out of range ;smallest : automatically reset the offset to the smallest offset; largest : automatically reset the offset to the largest offset;anything else: throw exception to the consumer
consumer.timeout.ms	-1	若在指定时间内没有消息消费，consumer将会抛出异常。
exclude.internal.topics	true	Whether messages from internal topics (such as offsets) should be exposed to the consumer.
zookeeper.session.timeout.ms	6000	ZooKeeper session timeout. If the consumer fails to heartbeat to ZooKeeper for this period of time it is considered dead and a rebalance will occur.
zookeeper.connection.timeout.ms	6000	The max time that the client waits while establishing a connection to zookeeper.
zookeeper.sync.time.ms	2000	How far a ZK follower can be behind a ZK leader

三 Kafka工作流程分析

3.1 认识Kafka生产者

生产者负责获取数据，比如flflume、自定义数据采集的脚本。生产者会监控一个目录负责把数据获取并发送给Kafka；

生产者集群是由多个进程组成，一个生产者作为一个独立的进程；

多个生产者发送的数据可以放到同一个topic的同一个分区；

一个生产者生产的数据可以放到多个topic中；

单个生产者具有数据分发的能力。

3.1.1 写入方式

producer 采用 push 模式将消息发布到 broker，每条消息都被 append 到 patition 中，属于顺序写磁盘（顺序写磁盘效率比随机写内存要高，保障 kafka 吞吐率）。

异步方式

批处理（一定数量的消息（64K大小的数据量）或者一定时间的延迟（每10ms发送一次））从kafka-0.8.2开始，producer不再区分同步（sync）和异步方式（async），所有的请求以异步方式发送，这样提升了客户端效率。producer请求会返回一个应答对象，包括偏移量或者错误信。异步地批量的发送消息到kafka broker节点，因而可以减少server端资源的开销。新的producer和所有的服务器网络通信都是异步地，在ack=-1模式下需要等待所有的replica副本完成复制时，可以大幅减少等待时间。producer采用推（push）模式将消息发布到broker，每条消息都被追加（append）到分区（patition）中，属于顺序写磁盘（顺序写磁盘效率比随机写内存要高，保障kafka吞吐率）。

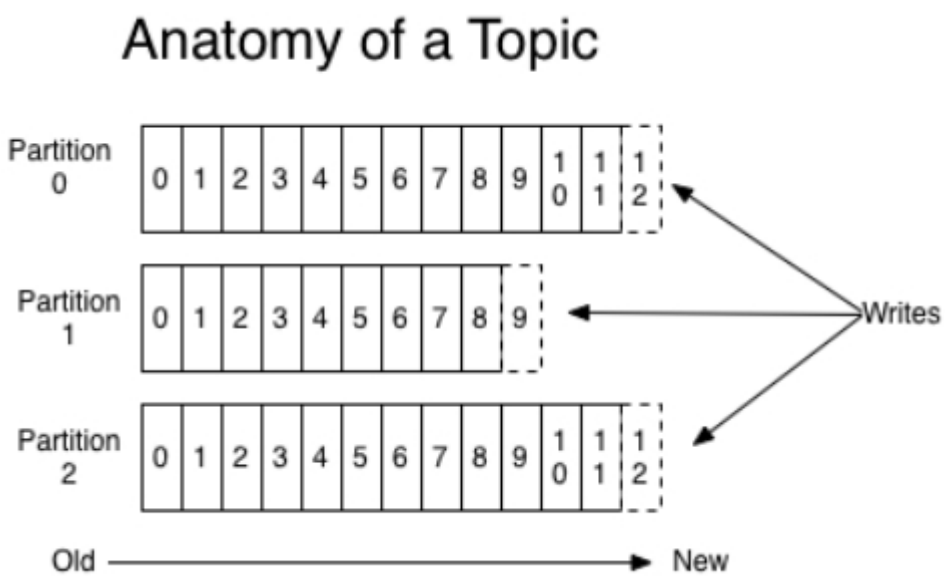
3.1.2 分区（Partition）

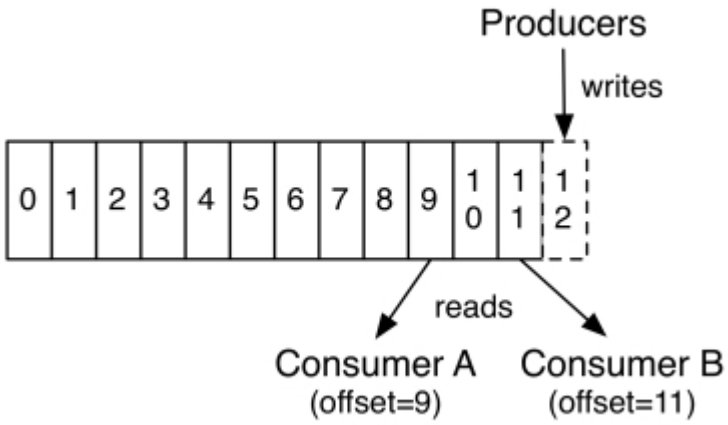
Kafka集群有多个消息代理服务器（broker-server）组成，发布到Kafka集群的每条消息都有一个类别，用主题（topic）来表示。通常，不同应用产生不同类型的数据，可以设置不同的主题。一个主题一般会有多个消息的订阅者，当生产者发布消息到某个主题时，订阅了这个主题的消费者都可以接收到生成者写入的新消息。

Kafka集群为每个主题维护了分布式的分区（partition）日志文件，物理意义上可以把主题（topic）看作进行了分区的日志文件（partition log）。主题的每个分区都是一个有序的、不可变的记录序列，新的消息会不断追加到日志中。分区中的每条消息都会按照时间顺序分配到一个单调递增的顺序编号，叫做偏移量（offset），这个偏移量能够唯一地定位当前分区中的每一条消息。

消息发送时都被发送到一个topic，其本质就是一个目录，而topic是由一些Partition Logs(分区日志)组成，其组织结构如下图所示：

下图中的topic有3个分区，每个分区的偏移量都从0开始，不同分区之间的偏移量都是独立的，不会相互影响。





我们可以看到，每个Partition中的消息都是有序的，生产的消息被不断追加到Partition log上，其中的每一个消息都被赋予了一个唯一的offset值。

发布到Kafka主题的每条消息包括键值和时间戳。消息到达服务器端的指定分区后，都会分配到一个自增的偏移量。原始的消息内容和分配的偏移量以及其他一些元数据信息最后都会存储到分区日志文件中。消息的键也可以不用设置，这种情况下消息会均衡地分布到不同的分区。

1）分区的原因

（1）方便在集群中扩展，每个Partition可以通过调整以适应它所在的机器，而一个topic又可以有多个Partition组成，因此整个集群就可以适应任意大小的数据了；

（2）可以提高并发，因为可以以Partition为单位读写了。

传统消息系统在服务端保持消息的顺序，如果有多个消费者消费同一个消息队列，服务端会以消费存储的顺序依次发送给消费者。但由于消息是异步发送给消费者的，消息到达消费者的顺序可能是无序的，这就意味着在并行消费时，传统消息系统无法很好地保证消息被顺序处理。虽然我们可以设置一个专用的消费者只消费一个队列，以此来解决消息顺序的问题，但是这就使得消费处理无法真正执行。

Kafka比传统消息系统有更强的顺序性保证，它使用主题的分区作为消息处理的并行单元。Kafka以分区作为最小的粒度，将每个分区分配给消费者组中不同的而且是唯一的消费者，并确保一个分区只属于一个消费者，即这个消费者就是这个分区的唯一读取线程。那么，只要分区的消息是有序的，消费者处理的消息顺序就有保证。每个主题有多个分区，不同的消费者处理不同的分区，所以Kafka不仅保证了消息的有序性，也做到了消费者的负载均衡。

2）分区的原则

（1）指定了patition，则直接使用；

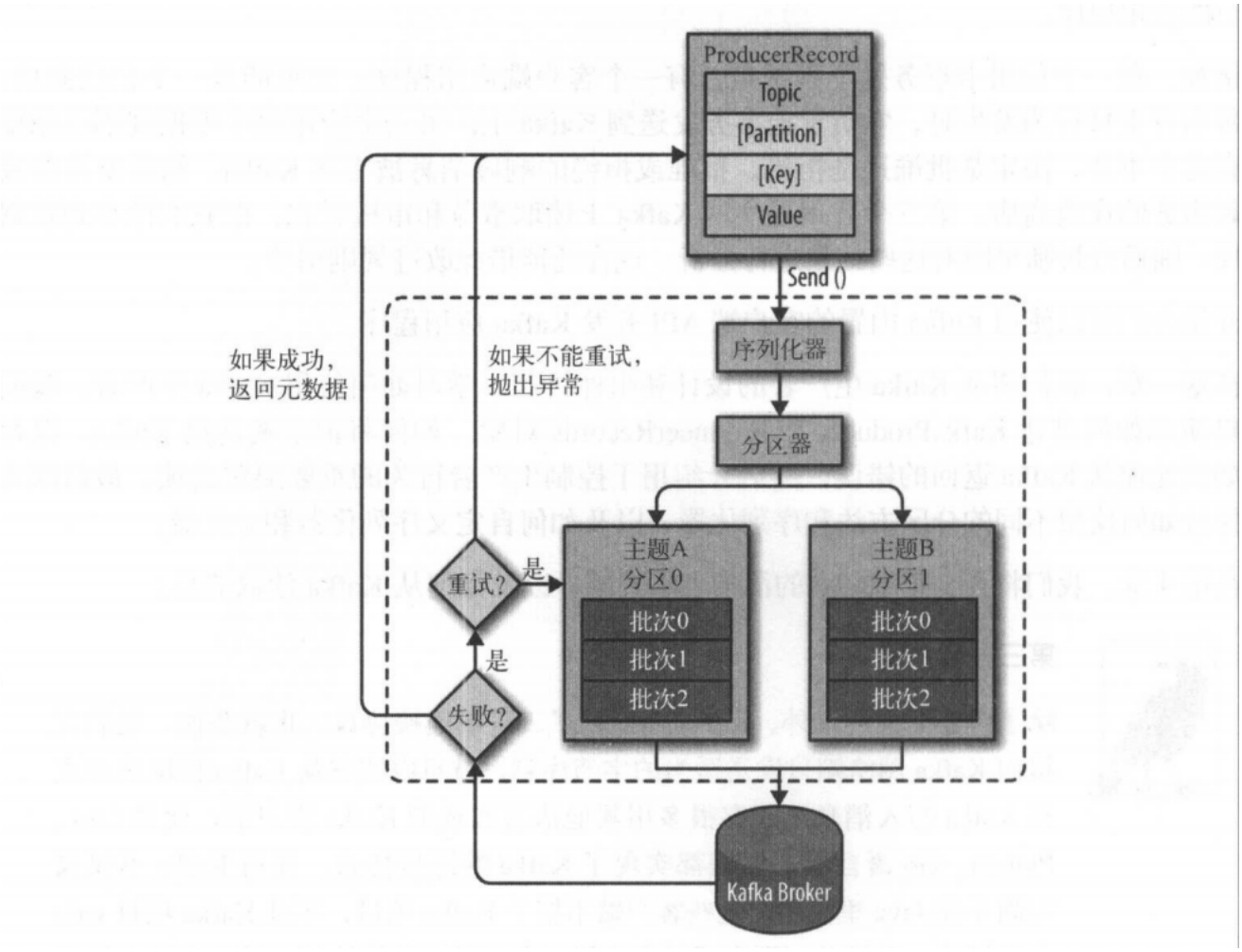
（2）未指定patition但指定key，通过对key的value进行hash出一个patition

（3）patition和key都未指定，使用轮询选出一个patition。

3.1.3 副本（Replication）

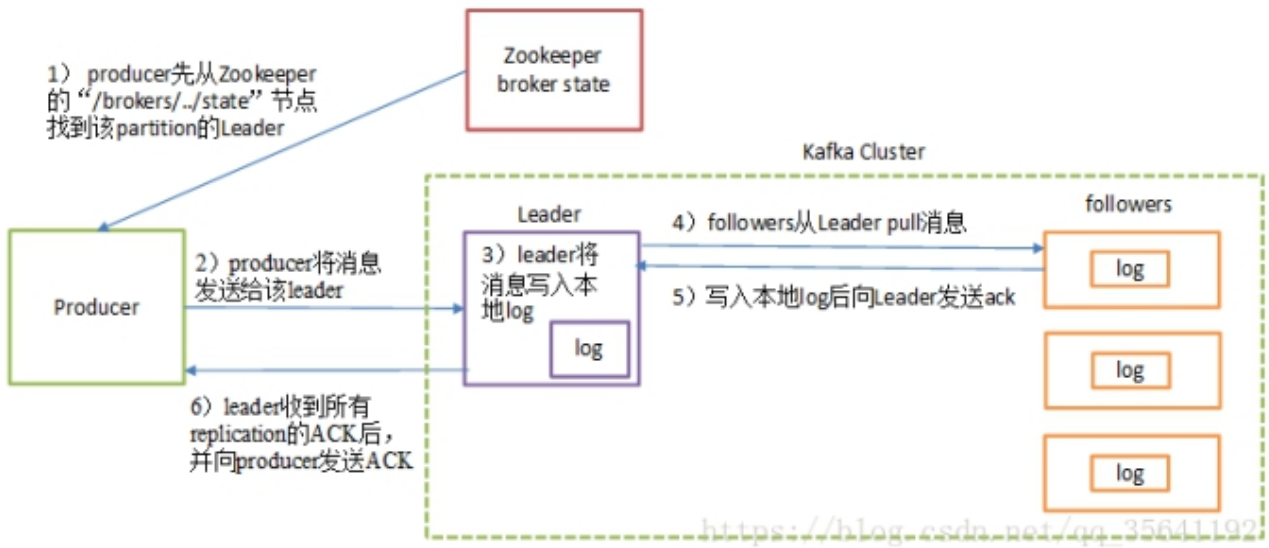
同一个partition可能会有多个replication（对应 server.properties 配置中的 default.replication.factor=N）。没有replication的情况下，一旦broker 宕机，其上所有 patition 的数据都不可被消费，同时producer也不能再将数据存于其上的patition。引入replication之后，同一个partition可能会有多个replication，而这时需要在这些replication之间选出一个leader，producer和consumer只与这个leader交互，其它replication作为follower从leader 中复制数据。

3.1.4 发送消息的流程



3.1.5 消息写入流程

producer写入消息流程如下：



1）producer先从zookeeper的 "/brokers/.../state"节点找到该partition的leader

2）producer将消息发送给该leader

3）leader将消息写入本地log

4）followers从leader pull消息，写入本地log后向leader发送ACK

5）leader收到所有ISR中的replication的ACK后，增加HW（high watermark，最后commit 的offset）并向producer发送ACK

ack = 0

ack = 1

ack =-1

kafka发确认的时机

只要leader写入成功就认为写入成功

leader以及所有的follower都写入成功，认为写入成功

不发确认机制

作为producer生产者

at least once ,只要没有返回确认信息，或者写入失败，都会进行消息的重发

at most once ，每条消息最多发送一次

exactly once ，保证每条发送且仅发送一次，而且是写入成功。

3.1.6 负载均衡和分区选择

kafka生产者把记录发送给topics，生产者需要抉择要把记录发送到topic的哪一个分区

根据记录是否有key,分区选择：

1，消息的key值为null，使用轮询方式；

2，消息有key值，按照哈希算法计算该消息key的hash值去选择分区；

Producer发送消息到broker时，会根据Paritition机制选择将其存储到哪一个Partition。如果Partition机制设置合理，所有消息可以均匀分布到不同的Partition里，这样就实现了负载均衡。如果一个Topic对应一个文件，那这个文件所在的机器I/O将会成为这个Topic的性能瓶颈，而有了Partition后，不同的消息可以并行写入不同broker的不同Partition里，极大的提高了吞吐率。可以在$KAFKA_HOME/confifig/server.properties中通过配置项num.partitions来指定新建Topic的默认Partition数量，也可在创建Topic时通过参数指定，同时也可以在Topic创建之后通过Kafka提供的工具修改。

其路由机制为：

指定了 patition，则直接使用；

未指定 patition 但指定 key，通过对 key 的 value 进行hash 选出一个 patition

patition 和 key 都未指定，使用轮询选出一个 patition。

3.1.7 创建生产者

```
private Properties kafkaProps = new properties();
kafkaProps.put ( "bootstrap.servers" , "broker1:9092,broker2:9092" ) ;
kafkaProps.put  ( "key.serializer","org.apache.kafka.common.serialization.StringSerializer");
kafkaProps.put  ( "value.serializer","org.apache.kafka.common.serialization.StringSerializer");
producer = new KafkaProducer<String,String>(kafkaProps)
```

相关属性

bootstrap.servers 该属性指定 broker 的地址清单，地址的格式为 host:port 。清单里不需要包含所有的broker 地址，生产者会从给定的 broker 里查找到其他 broker 的信息。不过建议至少要提供两个 broker 信息， 且其中一个若机，生产者仍然能够连接到集群上。 key.serializer broker 希望接收到的消息的键和值都是字节数组。生产者接口允许使用参数化类型，因此可以把 Java 对象作为键和值发送给 broker 。这样的代码具有良好的可读性，不过生产者需要知道如何把这些 Java 对象转换成字节数组。 key.serializer必须被设置为一个实现了 org.apache.kafka.common.serialization.Serializer 接口的类，生产者会使用这个类把键对象序列化成字节数组。Kafka 客户端默认提供了 ByteArraySerializer（这个只做很少的事情）、 StringSerializer和 IntegerSerializer,因此，如果你只使用常见的几种 Java 对象类型，那么就没必要实现自己的序列化器。要注意， key.serializer是必须设置的，就算你打算只发送值内容。 value. serializer 和key.serializer一样，指定的类会将值序列化。

3.1.8 发送消息的方式

发送并忘记 我们把消息发送给服务器，但并不关心它是否正常到达。大多数情况下，消息会正常到达，因为kafka是高可用的，而且生产者会自动尝试重发。不过，使用这种方式有时候也会丢失一些消息。 同步发送 使用 send（ ）方怯发送消息 它会返回Future对象，调用 get（ ）方法进行等待就可以知道悄息是否发送成功。 异步发送 我们调用 send（ ）方怯，并指定一个回调函数，服务器在返回响应时调用该函数。

3.2 Broker 保存消息

3.2.1 存储方式

物理上把topic分成一个或多个patition（对应 server.properties 中的num.partitions=3配置），每个patition物理上对应一个文件夹（该文件夹存储该patition的所有消息和索引文件），如下：

[1000phone@hadoop102 logs]$ ll

drwxrwxr-x. 2 1000phone 1000phone 4096 8月 6 14:37 first-0

drwxrwxr-x. 2 1000phone 1000phone 4096 8月 6 14:35 first-1

drwxrwxr-x. 2 1000phone 1000phone 4096 8月 6 14:37 first-2

[1000phone@hadoop102 logs]$ cd first-0

[1000phone@hadoop102 first-0]$ ll

-rw-rw-r--. 1 1000phone 1000phone 10485760 8月 6 14:33 00000000000000000000.index

-rw-rw-r--. 1 1000phone 1000phone 219 8月 6 15:07 00000000000000000000.log

-rw-rw-r--. 1 1000phone 1000phone 10485756 8月 6 14:33 00000000000000000000.timeindex

-rw-rw-r--. 1 1000phone 1000phone 8 8月 6 14:37 leader-epoch-checkpoint

3.2.2 存储策略

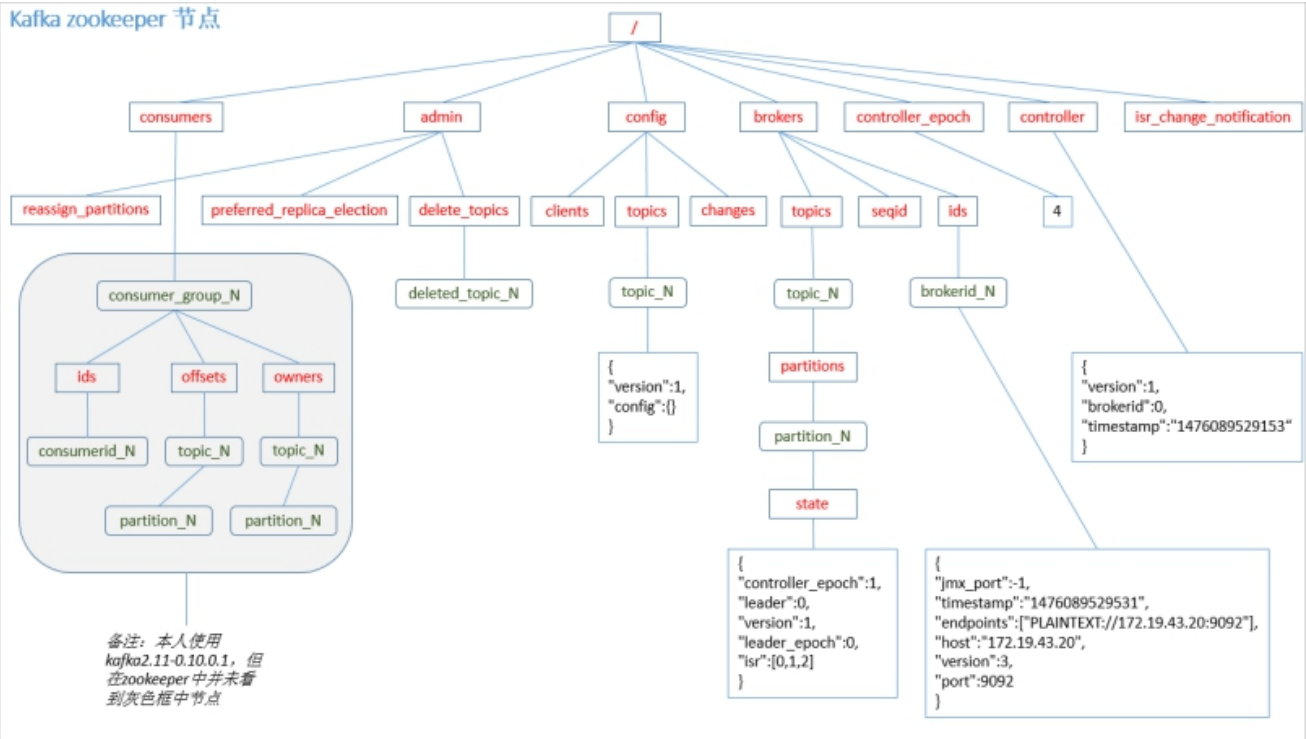
无论消息是否被消费，kafka都会保留所有消息。有两种策略可以删除旧数据：

1）基于时间：log.retention.hours=168

2）基于大小：log.retention.bytes=1073741824

需要注意的是，因为Kafka读取特定消息的时间复杂度为O(1)，即与文件大小无关，所以这里删除过期文件与提高 Kafka 性能无关。

3.2.3 Zookeeper存储结构



注意：producer不在zk中注册，消费者在zk中注册。

3.3 Kafka消费过程分析

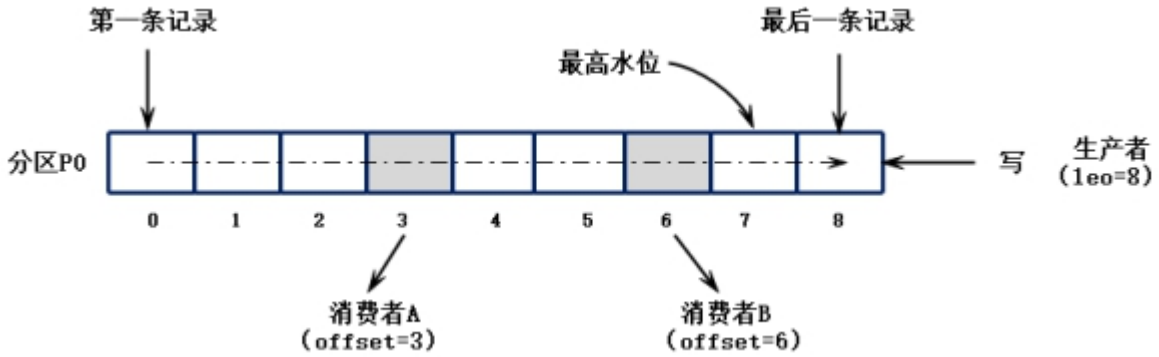
kafka提供了两套consumer API：高级Consumer API和低级API。

3.3.1 消费模型

消息由生产者发布到Kafka集群后，会被消费者消费。消息的消费模型有两种：推送模型（push）和拉取模型（pull）。

基于推送模型（push）的消息系统，由消息代理记录消费者的消费状态。消息代理在将消息推送到消费者后，标记这条消息为已消费，但这种方式无法很好地保证消息被处理。比如，消息代理把消息发送出去后，当消费进程挂掉或者由于网络原因没有收到这条消息时，就有可能造成消息丢失（因为消息代理已经把这条消息标记为已消费了，但实际上这条消息并没有被实际处理）。如果要保证消息被处理，消息代理发送完消息后，要设置状态为“已发送”，只有收到消费者的确认请求后才更新为“已消费”，这就需要消息代理中记录所有的消费状态，这种做法显然是不可取的。

Kafka采用拉取模型，由消费者自己记录消费状态，每个消费者互相独立地顺序读取每个分区的消息。如下图所示，有两个消费者（不同消费者组）拉取同一个主题的消息，消费者A的消费进度是3，消费者B的消费进度是6。消费者拉取的最大上限通过最高水位（watermark）控制，生产者最新写入的消息如果还没有达到备份数量，对消费者是不可见的。这种由消费者控制偏移量的优点是：消费者可以按照任意的顺序消费消息。比如，消费者可以重置到旧的偏移量，重新处理之前已经消费过的消息；或者直接跳到最近的位置，从当前的时刻开始消费。



在一些消息系统中，消息代理会在消息被消费之后立即删除消息。如果有不同类型的消费者订阅同一个主题，消息代理可能需要冗余地存储同一消息；或者等所有消费者都消费完才删除，这就需要消息代理跟踪每个消费者的消费状态，这种设计很大程度上限制了消息系统的整体吞吐量和处理延迟。Kafka的做法是生产者发布的所有消息会一致保存在Kafka集群中，不管消息有没有被消费。用户可以通过设置保留时间来清理过期的数据，比如，设置保留策略为两天。那么，在消息发布之后，它可以被不同的消费者消费，在两天之后，过期的消息就会自动清理掉。

3.3.2 高级API

1）高级API优点

高级API 写起来简单

不需要自行去管理offset，系统通过zookeeper自行管理。

不需要管理分区，副本等情况，.系统自动管理。

消费者断线会自动根据上一次记录在zookeeper中的offset去接着获取数据（默认设置1分钟更新一下zookeeper中存的offset）

可以使用group来区分对同一个topic 的不同程序访问分离开来（不同的group记录不同的offset，这样不同程序读取同一个topic才不会因为offset互相影响）

2）高级API缺点

不能自行控制offset（对于某些特殊需求来说）

不能细化控制如分区、副本、zk等

3.3.3 低级API

1）低级 API 优点

能够让开发者自己控制offset，想从哪里读取就从哪里读取。

自行控制连接分区，对分区自定义进行负载均衡

对zookeeper的依赖性降低（如：offset不一定非要靠zk存储，自行存储offset即可，比如存在文件或者内存中）

2）低级API缺点

太过复杂，需要自行控制offset，连接哪个分区，找到分区leader 等。

3.3.4 消费者组

消费者是以consumer group消费者组的方式工作，由一个或者多个消费者组成一个组，共同消费一个topic。每个分区在同一时间只能由group中的一个消费者读取，但是多个group可以同时消费这个partition。在图中，有一个由三个消费者组成的group，有一个消费者读取主题中的两个分区，另外两个分别读取一个分区。某个消费者读取某个分区，也可以叫做某个消费者是某个分区的拥有者。

在这种情况下，消费者可以通过水平扩展的方式同时读取大量的消息。另外，如果一个消费者失败了，那么其他的group成员会自动负载均衡读取之前失败的消费者读取的分区。

3.3.5 消费方式

consumer采用pull（拉）模式从broker中读取数据。

push（推）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broker决定的。它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consumer来不及处理消息，典型的表现就是拒绝服务以及网络拥塞。而pull模式则可以根据consumer的消费能力以适当的速率消费消息。

对于Kafka而言，pull模式更合适，它可简化broker的设计，consumer可自主控制消费消息的速率，同时consumer可以自己控制消费方式——即可批量消费也可逐条消费，同时还能选择不同的提交方式从而实现不同的传输语义。

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，一直等待数据到达。为了避免这种情况，我们在我们的拉请求中有参数，允许消费者请求在等待数据到达的“长轮询”中进行阻塞（并且可选地等待到给定的字节数，以确保大的传输大小）。

3.3.6 消费者组案例

1）需求：测试同一个消费者组中的消费者，同一时刻只能有一个消费者消费。

2）案例实操

（1）在hadoop102、hadoop103上修改/opt/module/kafka/config/consumer.properties配置文件中的group.id属性为任意组名。

[1000phone@hadoop103 config]$ vi consumer.properties

group.id=1000phone

（2）在hadoop102、hadoop103上分别启动消费者

[1000phone@hadoop102 kafka]$ bin/kafka-console-consumer.sh --zookeeper hadoop102:2181 --topic first --consumer.config config/consumer.properties

[1000phone@hadoop103 kafka]$ bin/kafka-console-consumer.sh --zookeeper hadoop102:2181 --topic first --consumer.config config/consumer.properties

（3）在hadoop104上启动生产者

[1000phone@hadoop104 kafka]$ bin/kafka-console-producer.sh --broker-list hadoop102:9092 --topic first

>hello world

（4）查看hadoop102和hadoop103的接收者。

同一时刻只有一个消费者接收到消息。

四 Kafka API实战

4.1 环境准备

1）在eclipse中创建一个java工程

2）在工程的根目录创建一个lib文件夹

3）解压kafka安装包，将安装包libs目录下的jar包拷贝到工程的lib目录下，并build path。

4）启动zk和kafka集群，在kafka集群中打开一个消费者

[1000phone@hadoop102 kafka]$ bin/kafka-console-consumer.sh --zookeeper hadoop102:2181 --topic first

4.2 Kafka生产者Java API

4.2.2 创建生产者（新API）

```
package com.1000phone.kafka;
import java.util.Properties;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerRecord;

public class NewProducer {

    public static void main(String[] args) {

        Properties props = new Properties();
        // Kafka服务端的主机名和端口号
        props.put("bootstrap.servers", "hadoop103:9092");
        // 等待所有副本节点的应答
        props.put("acks", "all");
        // 消息发送最大尝试次数
        props.put("retries", 0);
        // 一批消息处理大小
        props.put("batch.size", 16384);
        // 请求延时
        props.put("linger.ms", 1);
        // 发送缓存区内存大小
        props.put("buffer.memory", 33554432);
        // key序列化
        props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
        // value序列化
        props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

        KafkaProducer<String, String> producer = new KafkaProducer<>(props);
        for (int i = 0; i < 50; i++) {
```

```
            producer.send(new ProducerRecord<String, String>("first", Integer.toString(i),
"hello world-" + i));
        }

        producer.close();
    }
}
```

4.2.1消费者

```
import kafka.consumer.Consumer;
import kafka.consumer.ConsumerConfig;
import kafka.consumer.KafkaStream;
import kafka.javaapi.consumer.ConsumerConnector;
import kafka.message.MessageAndMetadata;

import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Properties;

public class ConsumerDemo {

    private static final String topic = "test2";
    private static final Integer threads = 2;

    public static void main(String[]args){
        Properties properties = new Properties();
        properties.put("zookeeper.connect","node1:2181,node2:2181,node3:2181");
        properties.put("group.id","xxx");
        properties.put("auto.offset.reset","smallest");
        ConsumerConfig consumerConfig = new ConsumerConfig(properties);

        //消费者连接器
        ConsumerConnector consumerConnector =
Consumer.createJavaConsumerConnector(consumerConfig);
        Map<String,Integer> topicMap = new HashMap<String,Integer>();
        //消费的topic,指定线程数
        topicMap.put(topic,threads);
        //获取数据
        Map<String, List<KafkaStream<byte[], byte[]>>> messageStreams =
consumerConnector.createMessageStreams(topicMap);
        List<KafkaStream<byte[], byte[]>> kafkaStreams = messageStreams.get(topic);
        //获取具体的消息
        for(final KafkaStream<byte[], byte[]> kafkaStream:kafkaStreams){
            new Thread(new Runnable() {
                @Override
                public void run() {
                    for(MessageAndMetadata<byte[],byte[]> mm:kafkaStream){
                       System.out.println(new String(mm.message()));


                    }
```

```
                }
            }).start();
        }


    }


}
```

4.2.3创建生产者带回调函数（新API）

```
package com.1000phone.kafka;
import java.util.Properties;
import org.apache.kafka.clients.producer.Callback;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.producer.RecordMetadata;

public class CallBackProducer {

    public static void main(String[] args) {

Properties props = new Properties();
        // Kafka服务端的主机名和端口号
        props.put("bootstrap.servers", "hadoop103:9092");
        // 等待所有副本节点的应答
        props.put("acks", "all");
        // 消息发送最大尝试次数
        props.put("retries", 0);
        // 一批消息处理大小
        props.put("batch.size", 16384);
        // 增加服务端请求延时
        props.put("linger.ms", 1);
// 发送缓存区内存大小
        props.put("buffer.memory", 33554432);
        // key序列化
        props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
        // value序列化
        props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

        KafkaProducer<String, String> kafkaProducer = new KafkaProducer<>(props);

        for (int i = 0; i < 50; i++) {


            kafkaProducer.send(new ProducerRecord<String, String>("first", "hello" + i), new
```

```
Callback() {

                @Override
                public void onCompletion(RecordMetadata metadata, Exception exception) {

                    if (metadata != null) {

                        System.out.println(metadata.partition() + "---" + metadata.offset());
                    }
                }
            });
        }

        kafkaProducer.close();
    }
}
```

4.2.4 自定义分区生产者

0）需求：将所有数据存储到topic的第0号分区上

1）定义一个类实现Partitioner接口，重写里面的方法（过时API）

```
package com.1000phone.kafka;
import java.util.Map;
import kafka.producer.Partitioner;

public class CustomPartitioner implements Partitioner {

    public CustomPartitioner() {
        super();
    }

    @Override
    public int partition(Object key, int numPartitions) {
        // 控制分区
        return 0;
    }
}
```

2）自定义分区（新API）

```
package com.1000phone.kafka;
import java.util.Map;
import org.apache.kafka.clients.producer.Partitioner;
import org.apache.kafka.common.Cluster;

public class CustomPartitioner implements Partitioner {

    @Override
```

```
    public void configure(Map<String, ?> configs) {

    }

    @Override
    public int partition(String topic, Object key, byte[] keyBytes, Object value, byte[]
valueBytes, Cluster cluster) {
        // 控制分区
        return 0;
    }

    @Override
    public void close() {

    }
}
```

3）在代码中调用

```
package com.1000phone.kafka;
import java.util.Properties;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerRecord;

public class PartitionerProducer {

    public static void main(String[] args) {

        Properties props = new Properties();
        // Kafka服务端的主机名和端口号
        props.put("bootstrap.servers", "hadoop103:9092");
        // 等待所有副本节点的应答
        props.put("acks", "all");
        // 消息发送最大尝试次数
        props.put("retries", 0);
        // 一批消息处理大小
        props.put("batch.size", 16384);
        // 增加服务端请求延时
        props.put("linger.ms", 1);
        // 发送缓存区内存大小
        props.put("buffer.memory", 33554432);
        // key序列化
        props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
        // value序列化
        props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
        // 自定义分区
        props.put("partitioner.class", "com.1000phone.kafka.CustomPartitioner");

        Producer<String, String> producer = new KafkaProducer<>(props);
        producer.send(new ProducerRecord<String, String>("first", "1", "1000phone"));
```

```
        producer.close();
    }
}
```

4）测试

（1）在hadoop102上监控/opt/module/kafka/logs/目录下first主题3个分区的log日志动态变化情况

[1000phone@hadoop102 first-0]$ tail -f 00000000000000000000.log

[1000phone@hadoop102 first-1]$ tail -f 00000000000000000000.log

[1000phone@hadoop102 first-2]$ tail -f 00000000000000000000.log

（2）发现数据都存储到指定的分区了。

4.3 Kafka消费者Java API

0）在控制台创建发送者

[1000phone@hadoop104 kafka]$ bin/kafka-console-producer.sh --broker-list hadoop102:9092 --topic first

>hello world

1）创建消费者（过时API）

```
package com.1000phone.kafka.consume;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Properties;
import kafka.consumer.Consumer;
import kafka.consumer.ConsumerConfig;
import kafka.consumer.ConsumerIterator;
import kafka.consumer.KafkaStream;
import kafka.javaapi.consumer.ConsumerConnector;

public class CustomConsumer {

    @SuppressWarnings("deprecation")
    public static void main(String[] args) {
        Properties properties = new Properties();

        properties.put("zookeeper.connect", "hadoop102:2181");
        properties.put("group.id", "g1");
        properties.put("zookeeper.session.timeout.ms", "500");
        properties.put("zookeeper.sync.time.ms", "250");
        properties.put("auto.commit.interval.ms", "1000");

        // 创建消费者连接器
        ConsumerConnector consumer = Consumer.createJavaConsumerConnector(new
ConsumerConfig(properties));
```

```
        HashMap<String, Integer> topicCount = new HashMap<>();
        topicCount.put("first", 1);

        Map<String, List<KafkaStream<byte[], byte[]>>> consumerMap =
consumer.createMessageStreams(topicCount);

        KafkaStream<byte[], byte[]> stream = consumerMap.get("first").get(0);

        ConsumerIterator<byte[], byte[]> it = stream.iterator();

        while (it.hasNext()) {
            System.out.println(new String(it.next().message()));
        }
    }
}
```

五 Kafka producer拦截器(interceptor)（扩展）

5.1 拦截器原理

Producer拦截器(interceptor)是在Kafka 0.10版本被引入的，主要用于实现clients端的定制化控制逻辑。

对于producer而言，interceptor使得用户在消息发送前以及producer回调逻辑前有机会对消息做一些定制化需求，比如修改消息等。同时，producer允许用户指定多个interceptor按序作用于同一条消息从而形成一个拦截链(interceptor chain)。Intercetpor的实现接口是org.apache.kafka.clients.producer.ProducerInterceptor，其定义的方法包括：

（1）configure(configs)

获取配置信息和初始化数据时调用。

（2）onSend(ProducerRecord)：

该方法封装进KafkaProducer.send方法中，即它运行在用户主线程中。Producer确保在消息被序列化以及计算分区前调用该方法。用户可以在该方法中对消息做任何操作，但最好保证不要修改消息所属的topic和分区，否则会影响目标分区的计算

（3）onAcknowledgement(RecordMetadata, Exception)：

该方法会在消息被应答或消息发送失败时调用，并且通常都是在producer回调逻辑触发之前。onAcknowledgement运行在producer的IO线程中，因此不要在该方法中放入很重的逻辑，否则会拖慢producer的消息发送效率

（4）close：

关闭interceptor，主要用于执行一些资源清理工作

如前所述，interceptor可能被运行在多个线程中，因此在具体实现时用户需要自行确保线程安全。另外倘若指定了多个interceptor，则producer将按照指定顺序调用它们，并仅仅是捕获每个interceptor可能抛出的异常记录到错误日志中而非在向上传递。这在使用过程中要特别留意。

5.2 拦截器案例

参照：https://cwiki.apache.org/confluence/display/KAFKA/KIP-42%3A+Add+Producer+and+Consumer+Interceptors

六 kafka Streams

6.1 概述

6.1.1 Kafka Streams

Kafka Stream是Apache Kafka从0.10版本引入的一个新Feature。它是提供了对存储于Kafka内的数据进行流式处理和分析的功能。

6.1.2 Kafka Streams特点

Kafka Stream的特点如下：

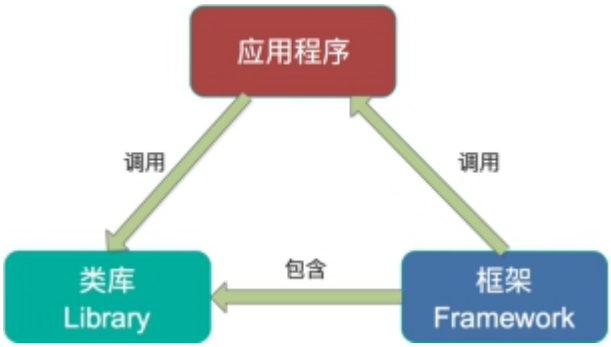
- Kafka Stream提供了一个非常简单而轻量的客户端库
- 除了Kafka外，无任何外部依赖
- 充分利用Kafka分区机制实现水平扩展和顺序性保证
- 通过可容错的state store实现高效的状态操作（如windowed join和aggregation）
- 支持exactly once语义
- 提供记录级的处理能力，从而实现毫秒级的低延迟
- 支持基于事件时间的窗口操作，并且可处理晚到的数据（late arrival of records）
- 同时提供底层的处理原语Processor（类似于Storm的spout和bolt），以及高层抽象的DSL（类似于Spark的map/group/reduce）

6.1.3 为什么要有Kafka Stream

当前已经有非常多的流式处理系统，最知名且应用最多的开源流式处理系统有Spark Streaming和Apache Storm。Apache Storm发展多年，应用广泛，提供记录级别的处理能力，当前也支持SQL on Stream。而Spark Streaming基于Apache Spark，可以非常方便与图计算，SQL处理等集成，功能强大，对于熟悉其它Spark应用开发的用户而言使用门槛低。另外，目前主流的Hadoop发行版，如Cloudera和Hortonworks，都集成了Apache Storm和Apache Spark，使得部署更容易。

既然Apache Spark与Apache Storm拥用如此多的优势，那为何还需要Kafka Stream呢？主要有如下原因。

第一，Spark和Storm都是流式处理框架，而Kafka Stream提供的是一个基于Kafka的流式处理类库。框架要求开发者按照特定的方式去开发逻辑部分，供框架调用。开发者很难了解框架的具体运行方式，从而使得调试成本高，并且使用受限。而Kafka Stream作为流式处理类库，直接提供具体的类给开发者调用，整个应用的运行方式主要由开发者控制，方便使用和调试。



第二，虽然Cloudera与Hortonworks方便了Storm和Spark的部署，但是这些框架的部署仍然相对复杂。而Kafka Stream作为类库，可以非常方便的嵌入应用程序中，它对应用的打包和部署基本没有任何要求。

第三，就流式处理系统而言，基本都支持Kafka作为数据源。例如Storm具有专门的kafka-spout，而Spark也提供专门的spark-streaming-kafka模块。事实上，Kafka基本上是主流的流式处理系统的标准数据源。换言之，大部分流式系统中都已部署了Kafka，此时使用Kafka Stream的成本非常低。

第四，使用Storm或Spark Streaming时，需要为框架本身的进程预留资源，如Storm的supervisor和Spark on YARN的node manager。即使对于应用实例而言，框架本身也会占用部分资源，如Spark Streaming需要为shuffle和storage预留内存。但是Kafka作为类库不占用系统资源。

第五，由于Kafka本身提供数据持久化，因此Kafka Stream提供滚动部署和滚动升级以及重新计算的能力。

第六，由于Kafka Consumer Rebalance机制，Kafka Stream可以在线动态调整并行度。

6.2 Kafka Stream案例

可参照官网

七 常见问题

7.1 Leader的选择

Kafka的核心是日志文件，日志文件在集群中的同步是分布式数据系统最基础的要素。如果leaders永远不会down的话我们就不需要followers了！一旦leader down掉了，需要在followers中选择一个新的leader.但是followers本身有可能延时太久或者crash，所以必须选择高质量的follower作为leader.必须保证，一旦一个消息被提交了，但是leader down掉了，新选出的leader必须可以提供这条消息。大部分的分布式系统采用了多数投票法则选择新的leader,对于多数投票法则，就是根据所有副本节点的状况动态的选择最适合的作为leader.Kafka并不是使用这种方法。Kafka动态维护了一个同步状态的副本的集合（a set of in-sync replicas），简称ISR，在这个集合中的节点都是和leader保持高度一致的，任何一条消息必须被这个集合中的每个节点读取并追加到日志中了，才会通知外部这个消息已经被提交了。因此这个集合中的任何一个节点随时都可以被选为leader.ISR在ZooKeeper中维护。ISR中有f+1个节点，就可以允许在f个节点down掉的情况下不会丢失消息并正常提供服。ISR的成员是动态的，如果一个节点被淘汰了，当它重新达到“同步中”的状态时，他可以重新加入ISR.这种leader的选择方式是非常快速的，适合kafka的应用场景。

如果所有节点都down掉了怎么办？

Kafka对于数据不会丢失的保证，是基于至少一个节点是存活的，一旦所有节点都down了，这个就不能保证了。实际应用中，当所有的副本都down掉时，必须及时作出反应。

可以有以下两种选择:

等待ISR(中断服务程序)中的任何一个节点恢复并担任leader。

选择所有节点中（不只是ISR）第一个恢复的节点作为leader.这是一个在可用性和连续性之间的权衡。

如果等待ISR中的节点恢复，一旦ISR中的节点起不起来或者数据丢失了，那集群就永远恢复不了了。如果等待ISR以外的节点恢复，这个节点的数据就会被作为线上数据，有可能和真实的数据有所出入，因为有些数据它可能还没同步到。Kafka目前选择了第二种策略，在未来的版本中将使这个策略的选择可配置，可以根据场景灵活的选择。

7.2 broker的数量

Kafka 集群需要多少个 broker 决于以下几个因素。

1)系统的吞吐量

2）磁盘的存储能力

首先，需要多少磁盘空间来保留数据，以及单个 broker 有多少空间可用。如果整个集群需要保留 lOTB 的数据，每个broker 可以存储 TB ，那么至少需要 broker 。如果启用了数据复制，那么至少还需要一倍的空间。 第二个要考虑的因素是集群处理请求的能力。这通常与网络接口处理客户端流量的能力有关，特别是当有多个消费者存在或者在数据保留期间流量发生波动（比如高峰时段的流量爆发）时。如果单个 broker 的网络接口在高峰时段可以达到 的使用量，并且有两消费者，那么消费者就无法保持峰值，除非有两个 broker 。如果集群启用了复制功能，就要把这个额外的消费者考虑在内。因磁盘吞吐量低和系 内存不足造成的性能问题，也可以通过扩展多个 broker 来解决。

7.3 如何设置生存周期

Kafka 日志消息保存时间总结

Kafka 日志实际上是以日志的方式默认保存在/kafka-logs文件夹中的。虽然默认有7天清除的机制，但是在数据量大，而磁盘容量不足的情况下，经常出现无法写入的情况。下面是相关参数的调整：

日志刷新策略

Kafka的日志实际上开始是在缓存中的，然后根据策略定期一批一批写入到日志文件中去，以提高吞吐率。

属性名	含义	默认值
log.flush.interval.messages	消息达到多少条时将数据写入到日志文件	9223372036854775807
log.flush.interval.ms	一条消息在内存时长当达到该时间时，强制执行一次flush，不设置的话，等同log.flush.scheduler.interval.ms	null
log.flush.scheduler.interval.ms	周期性检查，是否需要将信息flush	9223372036854775807

日志保存清理策略

属性名	含义	默认值
log.cleanup.policy	日志清理保存的策略只有delete和compact两种	delete
log.retention.hours	日志保存的时间，可以选择hours,minutes和ms	168(7day)
log.retention.bytes	删除前日志文件允许保存的最大值	-1
log.segment.delete.delay.ms	日志文件被真正删除前的保留时间	60000
log.retention.check.interval.ms	周期性检查是否有日志符合删除的条件（新版本使用）	300000

这里特别说明一下，日志的真正清除时间。当删除的条件满足以后，日志将被“删除”，但是这里的删除其实只是将该日志进行了“delete”标注，文件只是无法被索引到了而已。但是文件本身，仍然是存在的，只有当过了log.segment.delete.delay.ms 这个时间以后，文件才会被真正的从文件系统中删除。

7.4 kafka文件存储机制

分析过程分为以下4个步骤：

1. topic中partition存储分布
2. partiton中文件存储方式
3. partiton中segment文件存储结构
4. 在partition中如何通过offset查找message

通过上述4过程详细分析，我们就可以清楚认识到kafka文件存储机制的奥秘。

7.4.1 topic中partition的存储分布

假设实验环境中Kafka集群只有一个broker，xxx/message-folder为数据文件存储根目录，在Kafka broker中server.properties文件配置(参数log.dirs=xxx/message-folder)，例如创建2个topic名称分别为report_push、launch_info, partitions数量都为partitions=4 存储路径和目录规则为： xxx/message-folder

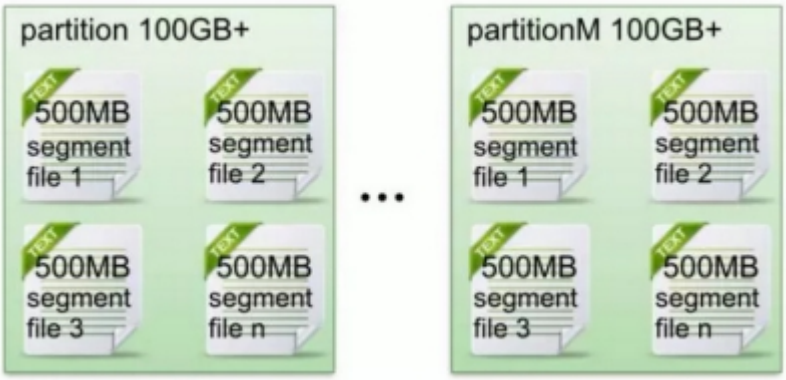
```
|--report_push-0
    |--report_push-1
    |--report_push-2
    |--report_push-3
    |--launch_info-0
    |--launch_info-1
    |--launch_info-2
    |--launch_info-3
```

在Kafka文件存储中，同一个topic下有多个不同partition，每个partition为一个目录，partiton命名规则为topic名称+有序序号，第一个partiton序号从0开始，序号最大值为partitions数量减1。

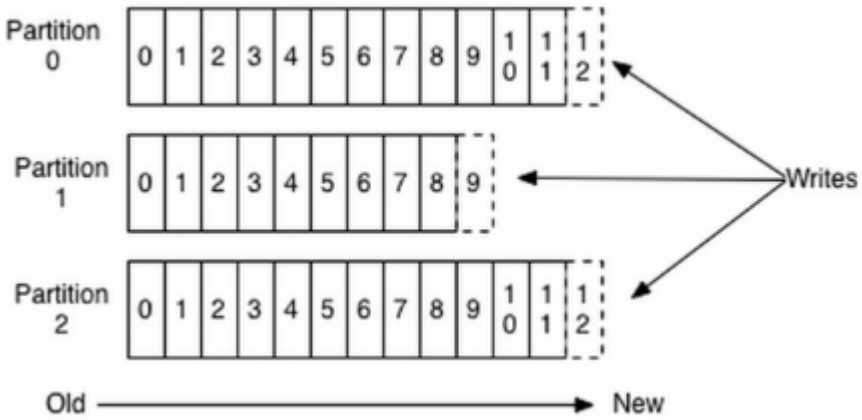
7.4.2 partiton中文件存储方式

在Kafka文件存储中，同一个topic下有多个不同partition，每个partition为一个分区，partiton命名规则为topic名称+有序序号，第一个partiton序号从0开始，序号最大值为partitions数量减1。

每个partion(分区)相当于一个巨型文件被平均分配到多个大小相等segment(段)数据文件中。**但每个段segment file消息数量不一定相等**，这种特性方便old segment file快速被删除。默认保留7天的数据。



每个partiton只需要支持顺序读写就行了，segment文件生命周期由服务端配置参数决定。（什么时候创建，什么时候删除）



数据有序的讨论？

一个partition的数据是否是有序的？ 间隔性有序，不连续

针对一个topic里面的数据，只能做到partition内部有序，不能做到全局有序。

特别加入消费者的场景后，如何保证消费者消费的数据全局有序的？伪命题。

只有一种情况下才能保证全局有序？就是只有一个partition。

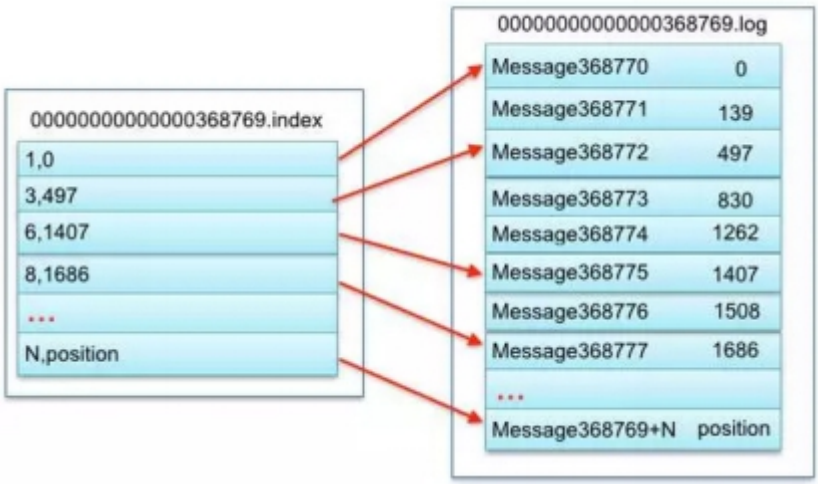
7.4.3 kafka分区中的Segment

Segment file组成：由2大部分组成，分别为index file和data file，此2个文件一一对应，成对出现，后缀".index"和“.log”分别表示为segment索引文件、数据文件。

00000000000000000000.index
00000000000000000000.log
00000000000000368769.index
00000000000000368769.log
00000000000000737337.index
00000000000000737337.log
00000000000001105814.index
00000000000001105814.log

Segment文件命名规则：partion全局的第一个segment从0开始，后续每个segment文件名为上一个segment文件最后一条消息的offset值。数值最大为64位long大小，19位数字字符长度，没有数字用0填充。

索引文件存储大量元数据，数据文件存储大量消息，索引文件中元数据指向对应数据文件中message的物理偏移地址。



3，497：当前log文件中的第几条信息，存放在磁盘上的那个地方

上述图中索引文件存储大量元数据，数据文件存储大量消息，索引文件中元数据指向对应数据文件中message的物理偏移地址。

其中以索引文件中元数据3,497为例，依次在数据文件中表示第3个message(在全局partiton表示第368772个message)、以及该消息的物理偏移地址为497。

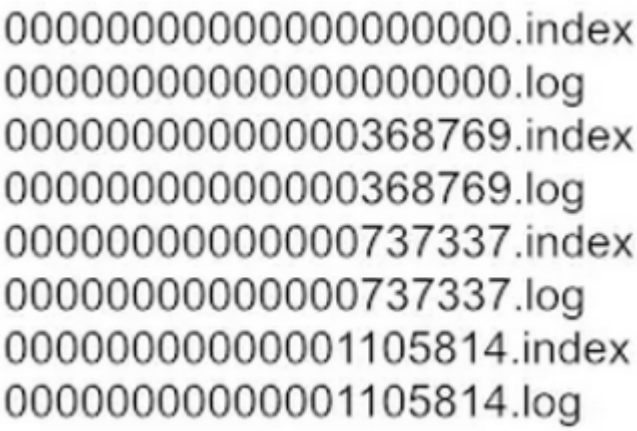
segment data file由许多message组成， qq物理结构如下：

关键字	解释说明
8 byte offset	在parition(分区)内的每条消息都有一个有序的id号，这个id号被称为偏移(offset),它可以唯一确定每条消息在parition(分区)内的位置。即offset表示partiion的第多少message
4 byte message size	message大小
4 byte CRC32	用crc32校验message
1 byte “magic"	表示本次发布Kafka服务程序协议版本号
1 byte “attributes"	表示为独立版本、或标识压缩类型、或编码类型。
4 byte key length	表示key的长度,当key为-1时，K byte key字段不填
K byte key	可选
value bytes payload	表示实际消息数据。

7.4.4 kafka怎样查找消息

元数据里维护每一个组对应每一个分区消费的offset

读取offset=368776的message，需要通过下面2个步骤查找。



1、查找segment file

00000000000000000000.index表示最开始的文件，起始偏移量(offset)为1

00000000000000368769.index的消息量起始偏移量为368770 = 368769 + 1

00000000000000737337.index的起始偏移量为737338=737337 + 1

其他后续文件依次类推。

以起始偏移量命名并排序这些文件，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。当offset=368776时定位到00000000000000368769.index和对应log文件。

2、通过segment file查找message

当offset=368776时，依次定位到00000000000000368769.index的元数据物理位置和00000000000000368769.log的物理偏移地址

然后再通过00000000000000368769.log顺序查找直到offset=368776为止。

7.5 kafka是怎样做到消息快速存储的

不同于Redis和MemcacheQ等内存消息队列，Kafka的设计是把所有的Message都要写入速度低容量大的硬盘，以此来换取更强的存储能力。实际上，Kafka使用硬盘并没有带来过多的性能损失，“规规矩矩”的抄了一条“近道”。

首先，说“规规矩矩”是因为Kafka在磁盘上只做Sequence I/O，由于消息系统读写的特殊性，这并不存在什么问题。关于磁盘I/O的性能，引用一组Kafka官方给出的测试数据(Raid-5，7200rpm)：

Sequence I/O: 600MB/s

Random I/O: 100KB/s

所以通过只做Sequence I/O的限制，规避了磁盘访问速度低下对性能可能造成的影响。

接下来我们再聊一聊Kafka是如何“抄近道的”。

首先，Kafka重度依赖底层操作系统提供的PageCache功能。当上层有写操作时，操作系统只是将数据写入PageCache，同时标记Page属性为Dirty。

当读操作发生时，先从PageCache中查找，如果发生缺页才进行磁盘调度，最终返回需要的数据。实际上PageCache是把尽可能多的空闲内存都当做了磁盘缓存来使用。同时如果有其他进程申请内存，回收PageCache的代价又很小，所以现代的OS都支持PageCache。

使用PageCache功能同时可以避免在JVM内部缓存数据，JVM为我们提供了强大的GC能力，同时也引入了一些问题不适用与Kafka的设计。

如果在Heap内管理缓存，JVM的GC线程会频繁扫描Heap空间，带来不必要的开销。如果Heap过大，执行一次Full GC对系统的可用性来说将是极大的挑战。

所有在在JVM内的对象都不免带有一个Object Overhead(千万不可小视)，内存的有效空间利用率会因此降低。

所有的In-Process Cache在OS中都有一份同样的PageCache。所以通过将缓存只放在PageCache，可以至少让可用缓存空间翻倍。

如果Kafka重启，所有的In-Process Cache都会失效，而OS管理的PageCache依然可以继续使用。

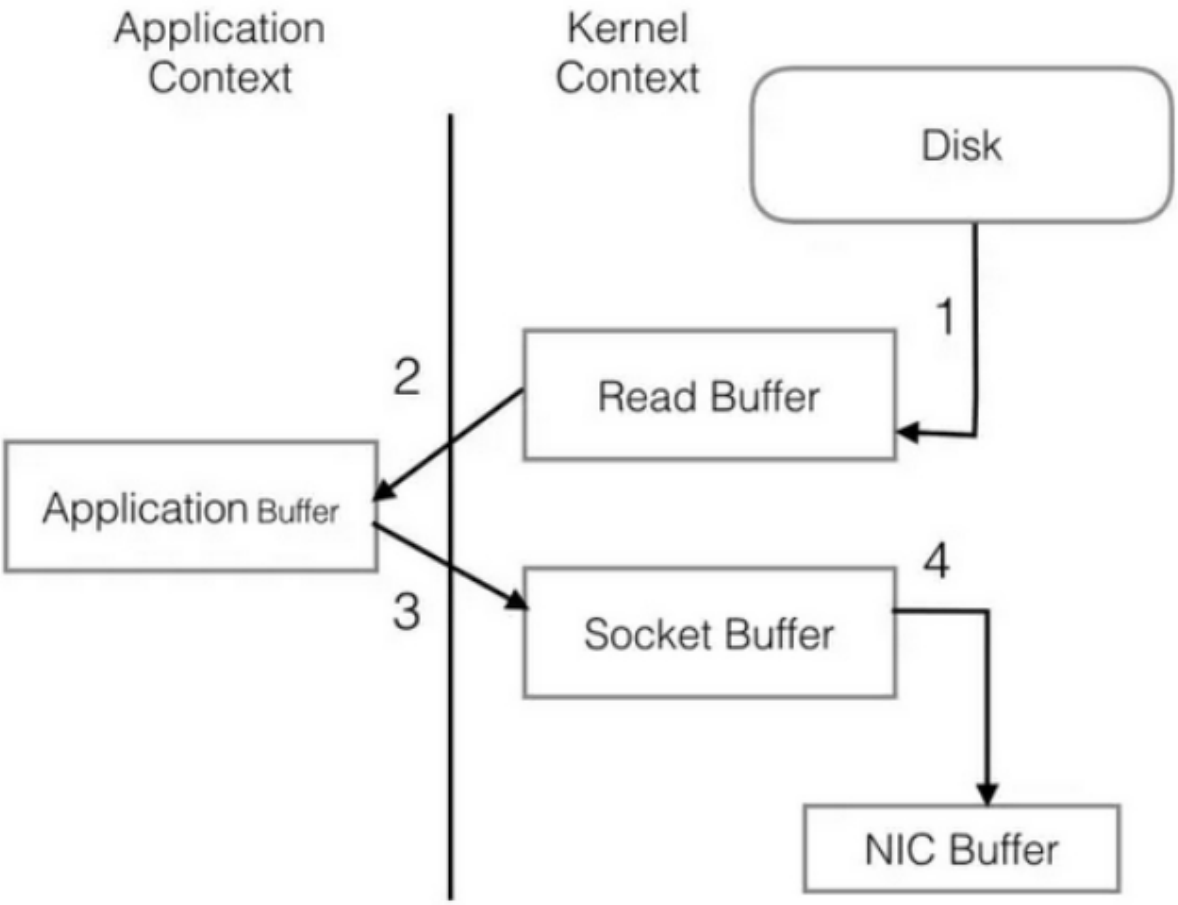
PageCache还只是第一步，Kafka为了进一步的优化性能还采用了Sendfile技术。在解释Sendfile之前，首先介绍一下传统的网络I/O操作流程，大体上分为以下4步。

OS 从硬盘把数据读到内核区的PageCache。

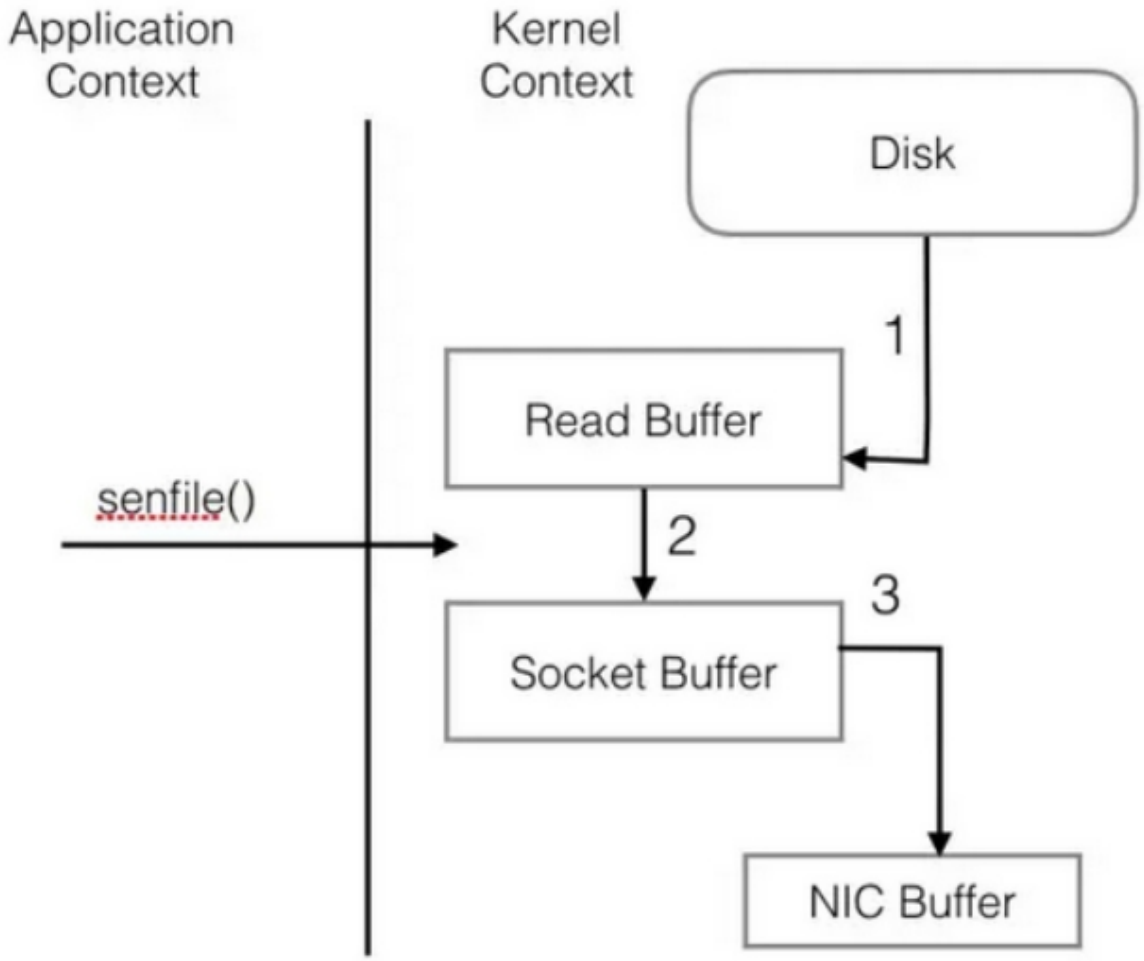
用户进程把数据从内核区Copy到用户区。

然后用户进程再把数据写入到Socket，数据流入内核区的Socket Buffer上。

OS 再把数据从Buffer中Copy到网卡的Buffer上，这样完成一次发送。



整个过程共经历两次Context Switch，四次System Call。同一份数据在内核Buffer与用户Buffer之间重复拷贝，效率低下。其中2、3两步没有必要，完全可以直接在内核区完成数据拷贝。这也正是Sendfile所解决的问题，经过Sendfile优化后，整个I/O过程就变成了下面这个样子。



通过以上的介绍不难看出，Kafka的设计初衷是尽一切努力在内存中完成数据交换，无论是对外作为一整个消息系统，或是内部同底层操作系统的交互。如果Producer和Consumer之间生产和消费进度上配合得当，完全可以实现数据交换零I/O。这也就是我为什么说Kafka使用“硬盘”并没有带来过多性能损失的原因。下面是我在生产环境中采到的一些指标。

(20 Brokers, 75 Partitions per Broker, 110k msg/s)

此时的集群只有写，没有读操作。10M/s左右的Send的流量是Partition之间进行Replicate而产生的。从recv和writ的速率比较可以看出，写盘是使用Asynchronous+Batch的方式，底层OS可能还会进行磁盘写顺序优化。而在有Read Request进来的时候分为两种情况，第一种是内存中完成数据交换。

Send流量从平均10M/s增加到了到平均60M/s，而磁盘Read只有不超过50KB/s。PageCache降低磁盘I/O效果非常明显。

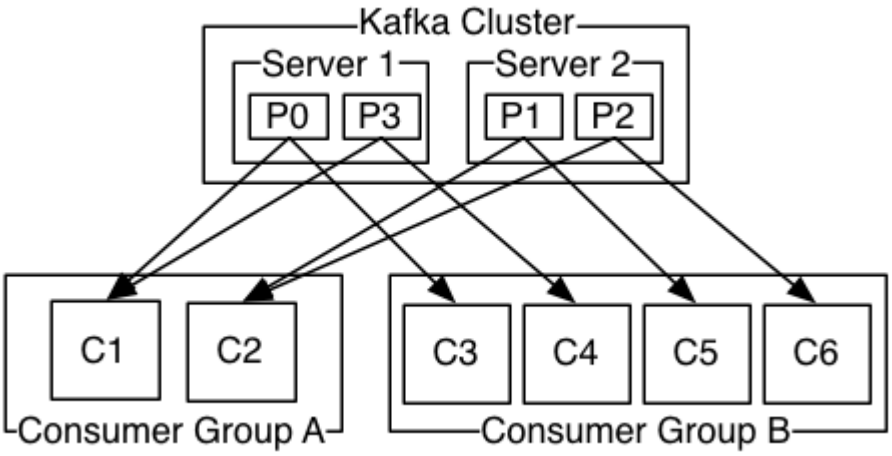
接下来是读一些收到了一段时间，已经从内存中被换出刷写到磁盘上的老数据。

其他指标还是老样子，而磁盘Read已经飚高到40+MB/s。此时全部的数据都已经是走硬盘了(对硬盘的顺序读取OS层会进行Prefill PageCache的优化)。依然没有任何性能问题。

7.6 kafka分区和消费者的关系

消费者以组的名义订阅主题，主题有多个分区，消费者组中有多个消费者实例，那么消费者实例和分区之前的对应关系是怎样的呢？

换句话说，就是组中的每一个消费者负责那些分区，这个分配关系是如何确定的呢？



同一时刻，一条消息只能被组中的一个消费者实例消费

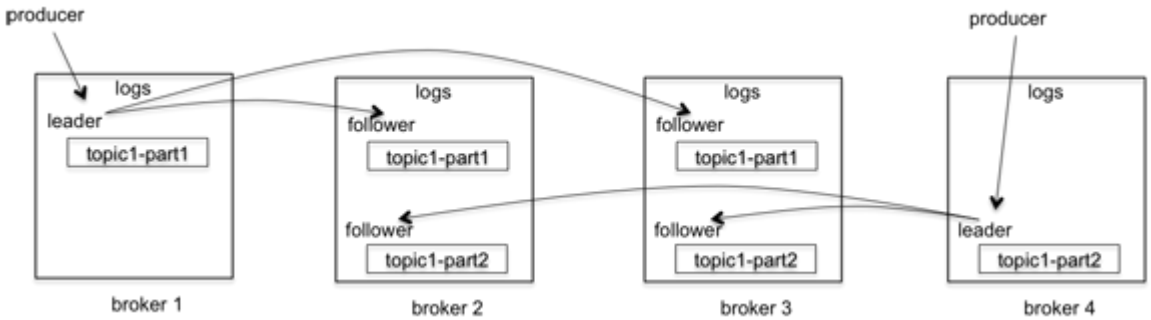
消费者组订阅这个主题，意味着主题下的所有分区都会被组中的消费者消费到，如果按照从属关系来说的话就是，主题下的每个分区只从属于组中的一个消费者，不可能出现组中的两个消费者负责同一个分区。

那么，问题来了。如果分区数大于或者等于组中的消费者实例数，那自然没有什么问题，无非一个消费者会负责多个分区，（当然，最理想的情况是二者数量相等，这样就相当于一个消费者负责一个分区）；但是，如果消费者实例的数量大于分区数，那么按照默认的策略（之所以强调默认策略是因为你也可以自定义策略），有一些消费者是多余的，一直接不到消息而处于空闲状态。

假设多个消费者负责同一个分区，那么会有什么问题呢？

我们知道，Kafka它在设计的时候就是要保证分区下消息的顺序，也就是说消息在一个分区中的顺序是怎样的，那么消费者在消费的时候看到的就是什么样的顺序，那么要做到这一点就首先要保证消息是由消费者主动拉取的（pull），其次还要保证一个分区只能由一个消费者负责。倘若，两个消费者负责同一个分区，那么就意味着两个消费者同时读取分区的消息，由于消费者自己可以控制读取消息的offset，就有可能C1才读到2，而C1读到1，C1还没处理完，C2已经读到3了，则会造成很多浪费，因为这就相当于多线程读取同一个消息，会造成消息处理的重复，且不能保证消息的顺序，这就跟主动推送（push）无异。

7.7 kafka的topic数据如何同步副本



kafka的复制是针对分区的。比如上图中有四个broker,一个topic,2个分区，复制因子是3。当producer发送一个消息的时候，它会选择一个分区，比如`topic1-part1`分区，将消息发送给这个分区的leader， broker2、broker3会拉取这个消息，一旦消息被拉取过来，slave会发送ack给master，这时候master才commit这个log。

这个过程中producer有两个选择：一是等所有的副本都拉取成功producer菜收到写入成功的response,二是等leader写入成功就得到成功的response。第一个中可以确保在异常情况下不丢消息，但是latency就下来了。后一种latency提高很多，但是一旦有异常情况，slave还没有来得及拉取到最新的消息leader就挂了，这种情况下就有可能丢消息了。

一个Broker既可能是一个分区的leader,也可能是另一个分区的slave，如上图所示。

kafka实际是保证在**足够多**的slave写入成功的情况下就认为消息写入成功，而不是全部写入成功。这是因为有可能一些节点网络不好，或者机器有问题hang住了，如果leader一直等着，那么所有后续的消息都堆积起来了， 所以kafka认为只要足够多的副本写入就可以饿。那么，怎么才认为是**足够多**呢？

Kafka引入了 **ISR**的概念。ISR是`in-sync replicas`的简写。ISR的副本保持和leader的同步，当然leader本身也在ISR中。初始状态所有的副本都处于ISR中，当一个消息发送给leader的时候，leader会等待ISR中所有的副本告诉它已经接收了这个消息，如果一个副本失败了，那么它会被移除ISR。下一条消息来的时候，leader就会将消息发送给当前的ISR中节点了。

同时，leader还维护这HW(high watermark),这是一个分区的最后一条消息的offset。HW会持续的将HW发送给slave，broker可以将它写入到磁盘中以便将来恢复。

当一个失败的副本重启的时候，它首先恢复磁盘中记录的HW，然后将它的消息truncate到HW这个offset。这是因为HW之后的消息不保证已经commit。这时它变成了一个slave， 从HW开始从Leader中同步数据，一旦追上leader，它就可以再加入到ISR中。

kafka使用Zookeeper实现leader选举。如果leader失败，controller会从ISR选出一个新的leader。leader 选举的时候可能会有数据丢失，但是committed的消息保证不会丢失。

7.8 kafka常见问题

producer集群：

1、生产者负责获取数据，比如flume、自定义数据采集的脚本。生产者会监控一个目录负责把数据获取并发送给Kafka

2、生产者集群是由多个进程组成，一个生产者作为一个独立的进程

3、多个生产者发送的数据是可以放到同一个topic的同一个分区的

4、一个生产者生产的数据可以放到多个topic里

5、单个生产者具有数据分发的能力

Kafka集群：

1、Kafka集群可以保存多种数据类型的数据，一种数据可以保存到一个topic里，一个kafka集群可以保存多个topic

2、每个topic里可以创建多个分区，分区的数量是在创建topic时指定的

3、每个分区的数据由多个Segment组成，一个Segment有index和data文件组成

4、一个topic的分区数据可以有多个备份，备份数也是在创建topic时指定，原始数据和备份数据不可以再同一个节点上

consumer集群：

1、消费者负责拉取数据进行消费，比如SparkStreaming实时的获取kafka的数据进行计算

2、一个ConsumerGroup称为Consumer集群

3、新增或减少consumer成员时会触发Consumer集群的负载均衡

4、ConsumerGroup（组）可以消费一个或多个分区的数据，反之，一个分区在同一时刻只能被一个consumer消费

5、consumer成员之间消费的数据各不相同，**而且数据不可以重复消费**

关于kafka涉及的几个重要问题：

1、Segment的概念？

一个分区被分成相同大小的段（segment），每个segment有多个索引文件（index）和数据文件（data）组成

2、数据是怎么保存到segment中的？（数据的存储机制是什么）

首先是Broker接收到数据以后，将数据放到操作系统的缓存里（pagecache），pagecache会尽可能多的使用空闲内存

使用sendfile技术尽可能多的减少操作系统和应用程序之间进行重复缓存

写入数据时使用顺序写入，写入的速度理论上可以达到600m/s

3、Consumer怎么解决负载均衡？

1、获取Consumer消费的起始分区号

2、计算出Consumer要消费的分区数量

3、用起始分区号的hashCode值模余分区数

4、数据的分发策略

Kafka默认调用自己的分区器（DefaultPartitioner），也可以自定义分区器，需要实现Partitioner接口，重写getPartition方法

5、Kafka怎么保证数据不丢失的？

Kafka接收数据后会根据创建的topic指定的备份数来存储数据，也就是多副本机制保证了数据的安全性

7.9 如何消费已经消费过的数据

1）消费数据消费过程，offset，修改之后，只是修改了当前组对应当前分区的offset值

2）一条消息只能被消费者组里的一个消费者消费

3）数据存储存储在kafka磁盘，默认7天

解决方案

1）使用其他的消费者组的消费者去消费数据

2）修改offset值（自己实现）

3）通过镜像kafka集群，从镜像集群去消费数据

consumer是底层采用的是一个阻塞队列，只要一有producer生产数据，那consumer就会将数据消费。当然这里会产生一个很严重的问题，如果你重启一消费者程序，那你连一条数据都抓不到，但是log文件中明明可以看到所有数据都好好的存在。换句话说，一旦你消费过这些数据，那你就无法再次用同一个groupid消费同一组数据了。

原因：消费者消费了数据并不从队列中移除，只是记录了offset偏移量。

同一个consumergroup的所有consumer合起来消费一个topic，并且他们每次消费的时候都会保存一个offset参数在zookeeper的root上。如果此时某个consumer挂了或者新增一个consumer进程，将会触发kafka的负载均衡，暂时性的重启所有consumer，重新分配哪个consumer去消费哪个partition，然后再继续通过保存在zookeeper上的offset参数继续读取数据。注意:offset保存的是consumer 组消费的消息偏移。

要消费同一组数据，可以采取如下措施：1)采用不同的group。2)通过一些配置，就可以将线上产生的数据同步到镜像中去，然后再由特定的集群区处理大批量的数据。

7.10 kafka保证数据一致性和可靠性

数据一致性保证一致性定义：若某条消息对client可见，那么即使Leader挂了，在新Leader上数据依然可以被读到
HW-HighWaterMark: client可以从Leader读到的最大msg offset，即对外可见的最大offset，
HW=max(replica.offset)对于Leader新收到的msg，client不能立刻消费，Leader会等待该消息被所有ISR中的replica同步后，更新HW，此时该消息才能被client消费，这样就保证了如果Leader fail，该消息仍然可以从新选举

的Leader中获取。对于来自内部Broker的读取请求，没有HW的限制。同时，Follower也会维护一份自己的HW，Folloer.HW = min(Leader.HW, Follower.offset)数据可靠性保证当Producer向Leader发送数据时，可以通过acks参数设置数据可靠性的级别

0: 不论写入是否成功，server不需要给Producer发送Response，如果发生异常，server会终止连接，触发Producer更新meta数据；

1: Leader写入成功后即发送Response，此种情况如果Leader fail，会丢失数据

-1: 等待所有ISR接收到消息后再给Producer发送Response，这是最强保证

7.11 kafka在高并发的情况下,如何避免消息丢失和消息重复?

首先对kafka进行限速， 其次启用重试机制，重试间隔时间设置长一些，最后Kafka设置acks=all，即需要相应的所有处于ISR的分区都确认收到该消息后，才算发送成功消息重复解决方案:消息可以使用唯一id标识 生产者（ack=all 代表至少成功发送一次) 消费者 （offset手动提交，业务逻辑成功处理后，提交offset） 落表（主键或者唯一索引的方式，避免重复数据） 业务逻辑处理（选择唯一主键存储到Redis或者mongdb中，先查询是否存在，若存在则不处理；若不存在，先插入Redis或Mongdb,再进行业务逻辑处理）

7.12 kafka怎么保证数据消费一次且仅消费一次

幂等producer：保证发送单个分区的消息只会发送一次，不会出现重复消息事务(transaction)：保证原子性地写入到多个分区，即写入到多个分区的消息要么全部成功，要么全部回滚流处理EOS：流处理本质上可看成是“读取-处理-写入”的管道。此EOS保证整个过程的操作是原子性。注意，这只适用于Kafka Streams

7.13 kafka到spark streaming怎么保证数据完整性，怎么保证数据不重复消费？

保证数据不丢失（at-least）spark RDD内部机制可以保证数据at-least语义。

Receiver方式开启WAL（预写日志），将从kafka中接受到的数据写入到日志文件中，所有数据从失败中可恢复。Direct方式依靠checkpoint机制来保证。

保证数据不重复（exactly-once）要保证数据不重复，即Exactly once语义。 - 幂等操作：重复执行不会产生问题，不需要做额外的工作即可保证数据不重复。 - 业务代码添加事务操作就是说针对每个partition的数据，产生一个uniqueId，只有这个partition的所有数据被完全消费，则算成功，否则算失效，要回滚。下次重复执行这个uniqueId时，如果已经被执行成功，则skip掉。

7.14 kafka的消费者高阶和低阶API有什么区别

kafka 提供了两套 consumer API：The high-level Consumer API和 The SimpleConsumer API

其中 high-level consumer API 提供了一个从 kafka 消费数据的高层抽象，而 SimpleConsumer API 则需要开发人员更多地关注细节。

The high-level consumer API：high-level consumer API 提供了 consumer group 的语义，一个消息只能被 group 内的一个 consumer 所消费，且 consumer 消费消息时不关注 offset，最后一个 offset 由 zookeeper 保存。

使用 high-level consumer API 可以是多线程的应用，应当注意：如果消费线程大于 patition 数量，则有些线程将收不到消息如果 patition 数量大于线程数，则有些线程多收到多个 patition 的消息如果一个线程消费多个 patition，则无法保证你收到的消息的顺序，而一个 patition 内的消息是有序的The SimpleConsumer API如果你想要对 patition 有更多的控制权，那就应该使用 SimpleConsumer API，比如：多次读取一个消息只消费一个

patition 中的部分消息使用事务来保证一个消息仅被消费一次但是使用此 API 时，partition、offset、broker、leader 等对你不再透明，需要自己去管理。你需要做大量的额外工作：必须在应用程序中跟踪 offset，从而确定下一条应该消费哪条消息应用程序需要通过程序获知每个 Partition 的 leader 是谁需要处理 leader 的变更

7.15 kafka的exactly-once（支持正好一次的消息传递）

幂等producer：保证发送单个分区的消息只会发送一次，不会出现重复消息

事务(transaction)：保证原子性地写入到多个分区，即写入到多个分区的消息要么全部成功，要么全部回滚

流处理EOS：流处理本质上可看成是“读取-处理-写入”的管道。此EOS保证整个过程的操作是原子性。注意，这只适用于Kafka Streams

7.16 如何保证从Kafka获取数据不丢失?

1.生产者数据的不丢失kafka的ack机制：在kafka发送数据的时候，每次发送消息都会有一个确认反馈机制，确保消息正常的能够被收到。

2.消费者数据的不丢失通过offset commit 来保证数据的不丢失，kafka自己记录了每次消费的offset数值，下次继续消费的时候，接着上次的offset进行消费即可。