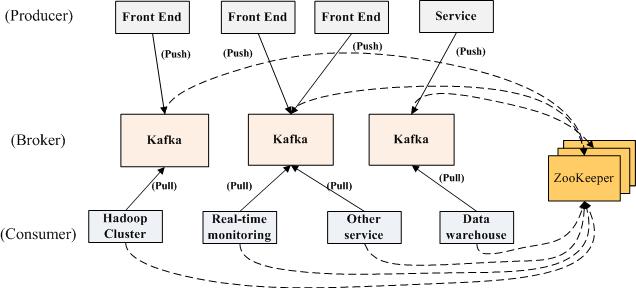
1. **Kafka是什么，主要应用在什么场景?**
2. **kafka是什么？**

Kafka是由LinkedIn开发的一个分布式基于发布/订阅的消息系统，使用Scala编写，它以可水平扩展和高吞吐率而被广泛使用。

1. **产生背景**

Kafka是一个消息系统，用作LinkedIn的活动流（Activity Stream）和运营数据处理管道（Pipeline）的基础。活动流数据是几乎所有站点在对其网站使用情况做报表时都要用到的数据中最常规的部分。活动数据包括页面访问量（Page View）、被查看内容方面的信息以及搜索情况等内容。这种数据通常的处理方式是先把各种活动以日志的形式写入某种文件，然后周期性地对这些文件进行统计分析。运营数据指的3是服务器的性能数据（CPU、IO使用率、请求时间、服务日志等等数据)。运营数据的统计方法种类繁多。

1. **基本架构图**



1. **基本概念解释**

**1）Broker**

Kafka集群包含一个或多个服务器，这种服务器被称为broker。broker端不维护数据的消费状态，提升了性能。直接使用磁盘进行存储，线性读写，速度快：避免了数据在JVM内存和系统内存之间的复制，减少耗性能的创建对象和垃圾回收。

**2）Producer**

负责发布消息到Kafka broke

**3）Consumer**

消息消费者，向Kafka broker读取消息的客户端，consumer从broker拉取(pull)数据并进行处理。

**4）Topic**

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）

**5）Partition**

Parition是物理上的概念，每个Topic包含一个或多个Partition.

**6）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）

**7）Topic & Partition**

Topic在逻辑上可以被认为是一个queue，每条消费都必须指定它的Topic，可以简单理解为必须指明把这条消息放进哪个queue里。为了使得Kafka的吞吐率可以线性提高，物理上把Topic分成一个或多个Partition，每个Partition在物理上对应一个文件夹，该文件夹下存储这个Partition的所有消息和索引文件。若创建topic1和topic2两个topic，且分别有13个和19个分区，则整个集群上会相应会生成共32个文件夹（本文所用集群共8个节点，此处topic1和topic2 replication-factor均为1）。

**5、适用场景**

**1、Messaging**

    对于一些常规的消息系统,kafka是个不错的选择;partitons/replication和容错,可以使kafka具有良好的扩展性和性能优势.不过到目前为止,我们应该很清楚认识到,kafka并没有提供JMS中的"事务性""消息传输担保(消息确认机制)""消息分组"等企业级特性;kafka只能使用作为"常规"的消息系统,在一定程度上,尚未确保消息的发送与接收绝对可靠(比如,消息重发,消息发送丢失等)

**2、Website activity tracking**

    kafka可以作为"网站活性跟踪"的最佳工具;可以将网页/用户操作等信息发送到kafka中.并实时监控,或者离线统计分析等

**3、Metrics**

         Kafka通常被用于可操作的监控数据。这包括从分布式应用程序来的聚合统计用来生产集中的运营数据提要。

**4、Log Aggregation**

kafka的特性决定它非常适合作为"日志收集中心";application可以将操作日志"批量""异步"的发送到kafka集群中,而不是保存在本地或者DB中;kafka可以批量提交消息/压缩消息等,这对producer端而言,几乎感觉不到性能的开支.此时consumer端可以使hadoop等其他系统化的存储和分析系统

1. **Kafka基本安装使用方法**

**1、 安装**

a. 首先安装JRE/JDK

Linux安装JDK

b. 下载kafka

进入下载页面：http://kafka.apache.org/downloads.html

选择Binary downloads下载 （Source download需要编译才能使用）

/bin 启动和停止命令等。

/config 配置文件

/libs 类库

c. 解压 kafka\_2.11-0.9.0.1.tgz

d. 修改配置

Kafka默认开启JVM压缩指针，但只是在64位的HotSpot VM受支持，如果安装了32位的HotSpot VM，需要修改/bin/kafka-run-class.sh文件

Shell代码

vi bin/kafka-run-class.sh

找到如下行：

Sh代码

KAFKA\_JVM\_PERFORMANCE\_OPTS="-server -XX:+UseCompressedOops -XX:+UseParNewGC -XX:+UseConcMarkSweepGC -XX:+CMSClassUnloadingEnabled -XX:+CMSScavengeBeforeRemark -XX:+DisableExplicitGC -Djava.awt.headless=true"

去除-XX:+UseCompressedOops参数

**2. 启动和停止**

启动Zookeeper server：

Shell代码

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

&是为了能退出命令行

启动Kafka server:

Shell代码

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

停止Kafka server

Shell代码

bin/kafka-server-stop.sh

停止Zookeeper server:

Shell代码

bin/zookeeper-server-stop.sh

**3. 单机连通性测试**

运行producer：

Shell代码

bin/kafka-console-producer.sh --broker-list localhost:9092 --topic test

早版本的Kafka，--broker-list localhost:9092需改为--zookeeper localhost:2181

运行consumer：

Shell代码

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

在producer端输入字符串并回车，查看consumer端是否显示。

**4. 分布式连通性测试**

Zookeeper Server, Kafka Server, Producer都放在服务器server1上，ip地址为10.10.73.58

Consumer放在服务器server2上，ip地址为10.10.73.53。

分别运行server1的producer和server2的consumer，

Shell代码

bin/kafka-console-producer.sh --broker-list 10.10.73.58:9092 --topic test

Shell代码

bin/kafka-console-consumer.sh --zookeeper 10.10.73.58:2181 --topic test --from-beginning

**三、Kafka线上环境集群部署及客户端应用**

1. ****Zookeeper集群构建****

我们有3个zk实例，分别为zk-0,zk-1,zk-2;如果你仅仅是测试使用，可以使用1个zk实例.（本示例基于分布式部署）

****1) zk-0****

调整配置文件：

clientPort=2181

dataDir=/opt/zookeeper-3.4.6/data

server.0=10.10.73.53:2888:3888

server.1=10.10.73.54:2888:3888

server.2=10.10.73.58:2888:3888

##只需要修改上述配置，其他配置保留默认值

启动zookeeper

./zkServer.sh start

**2) zk-1**

调整配置文件(其他配置和zk-0一只)：

clientPort=2182

##只需要修改上述配置，其他配置保留默认值

启动zookeeper

./zkServer.sh start

3) zk-2

调整配置文件(其他配置和zk-0一只)：

clientPort=2183

##只需要修改上述配置，其他配置保留默认值

启动zookeeper

./zkServer.sh start

**2. Kafka集群构建**

因为Broker配置文件涉及到zookeeper的相关约定，因此我们先展示broker配置文件.我们使用2个kafka broker来构建这个集群环境，分别为kafka-0,kafka-1.

**1) kafka-0(10.10.73.58)**

在config目录下修改配置文件为：

broker.id=0

port=9092

num.network.threads=2

num.io.threads=2

socket.send.buffer.bytes=1048576

socket.receive.buffer.bytes=1048576

socket.request.max.bytes=104857600

log.dir=/opt/env/kafka\_2.11-0.9.0.1/logs

num.partitions=2

log.flush.interval.messages=10000

log.flush.interval.ms=1000

log.retention.hours=168

#log.retention.bytes=1073741824

log.segment.bytes=536870912

##replication机制,让每个topic的partitions在kafka-cluster中备份2个

##用来提高cluster的容错能力..

default.replication.factor=1

log.cleanup.interval.mins=10

zookeeper.connect=10.10.73.53:2181,10.10.73.54:2181,10.10.73.58:2181

zookeeper.connection.timeout.ms=1000000

启动kafka broker：

> JMS\_PORT=9997 bin/kafka-server-start.sh config/server.properties &

因为zookeeper环境已经正常运行了，我们无需通过kafka来挂载启动zookeeper.如果你的一台机器上部署了多个kafka broker，你需要声明JMS\_PORT.

**2) kafka-1(10.10.73.53)**

broker.id=1

port=9092

##其他配置和kafka-0保持一致

然后和kafka-0一样执行打包命令，然后启动此broker.

> JMS\_PORT=9998 bin/kafka-server-start.sh config/server.properties &

仍然可以通过如下指令查看topic的"partition"/"replicas"的分布和存活情况.

**Kafka集群中broker.id=1参数的value不能重复**

**3、Java端的Producer**

kafka-producer.properties配置文件

##metadata.broker.list=127.0.0.1:9092,127.0.0.1:9093

##,127.0.0.1:9093

metadata.broker.list=10.10.73.58:9092,10.10.73.53:9092

producer.type=sync

compression.codec=0

serializer.class=kafka.serializer.StringEncoder

#batch.num.messages=100

Java客户端程序：

public class KafkaProducerClient {

private Producer<String, String> inner;

private String brokerList;//for metadata discovery,spring setter

private String location = "kafka/kafka-producer.properties";//spring setter

private String defaultTopic;//spring setter

public void setBrokerList(String brokerList) {

this.brokerList = brokerList;

}

public void setLocation(String location) {

this.location = location;

}

public void setDefaultTopic(String defaultTopic) {

this.defaultTopic = defaultTopic;

}

public KafkaProducerClient(){}

public void init() throws Exception {

Properties properties = new Properties();

properties.load(Thread.currentThread().getContextClassLoader().getResourceAsStream(location));

if(brokerList != null) {

properties.put("metadata.broker.list", brokerList);

}

ProducerConfig config = new ProducerConfig(properties);

inner = new Producer<String, String>(config);

}

public void send(String message){

send(defaultTopic,message);

}

public void send(Collection<String> messages){

send(defaultTopic,messages);

}

public void send(String topicName, String message) {

if (topicName == null || message == null) {

return;

}

KeyedMessage<String, String> km = new KeyedMessage<String, String>(topicName,message);

inner.send(km);

}

public void send(String topicName, Collection<String> messages) {

if (topicName == null || messages == null) {

return;

}

if (messages.isEmpty()) {

return;

}

List<KeyedMessage<String, String>> kms = new ArrayList<KeyedMessage<String, String>>();

int i= 0;

for (String entry : messages) {

KeyedMessage<String, String> km = new KeyedMessage<String, String>(topicName,entry);

kms.add(km);

i++;

if(i % 20 == 0){

inner.send(kms);

kms.clear();

}

}

if(!kms.isEmpty()){

inner.send(kms);

}

}

public void close() {

inner.close();

}

/\*\*

\* @param args

\*/

public static void main(String[] args) {

KafkaProducerClient producer = null;

try {

producer = new KafkaProducerClient();

producer.init();

//producer.setBrokerList("");

int i = 0;

while (true) {

producer.send("test-topic", "this is a sample" + i);

i++;

Thread.sleep(2000);

}

} catch (Exception e) {

e.printStackTrace();

} finally {

if (producer != null) {

producer.close();

}

}

}

}

1. **Java端的Consumer**

kafka-consumer.properties配置文件

##zookeeper.connect=127.0.0.1:2181,127.0.0.1:2182,127.0.0.1:2183

##,127.0.0.1:2182,127.0.0.1:2183

# timeout in ms for connecting to zookeeper

zookeeper.connect=10.10.73.53:2181,10.10.73.54:2181,10.10.73.58:2181

zookeeper.connectiontimeout.ms=100000

#consumer group id

group.id=test-group

#consumer timeout

#consumer.timeout.ms=5000

Java客户端程序：

public class KafkaConsumerClient {

private String groupid; //can be setting by spring

private String zkConnect;//can be setting by spring

private String location = "kafka/kafka-consumer.properties";//配置文件位置

private String topic;

private int partitionsNum;

private MessageExecutor executor; //message listener

private ExecutorService threadPool;

private ConsumerConnector connector;

private Charset charset = Charset.forName("utf8");

public void setGroupid(String groupid) {

this.groupid = groupid;

}

public void setZkConnect(String zkConnect) {

this.zkConnect = zkConnect;

}

public void setLocation(String location) {

this.location = location;

}

public void setTopic(String topic) {

this.topic = topic;

}

public void setPartitionsNum(int partitionsNum) {

this.partitionsNum = partitionsNum;

}

public void setExecutor(MessageExecutor executor) {

this.executor = executor;

}

public KafkaConsumerClient() {}

//init consumer,and start connection and listener

public void init() throws Exception {

if(executor == null){

throw new RuntimeException("KafkaConsumer,exectuor cant be null!");

}

Properties properties = new Properties();

properties.load(Thread.currentThread().getContextClassLoader().getResourceAsStream(location));

if(groupid != null){

properties.put("groupid", groupid);

}

if(zkConnect != null){

properties.put("zookeeper.connect", zkConnect);

}

ConsumerConfig config = new ConsumerConfig(properties);

connector = Consumer.createJavaConsumerConnector(config);

Map<String, Integer> topics = new HashMap<String, Integer>();

topics.put(topic, partitionsNum);

Map<String, List<KafkaStream<byte[], byte[]>>> streams = connector.createMessageStreams(topics);

List<KafkaStream<byte[], byte[]>> partitions = streams.get(topic);

threadPool = Executors.newFixedThreadPool(partitionsNum \* 2);

//start

for (KafkaStream<byte[], byte[]> partition : partitions) {

threadPool.execute(new MessageRunner(partition));

}

}

public void close() {

try {

threadPool.shutdownNow();

} catch (Exception e) {

//

} finally {

connector.shutdown();

}

}

class MessageRunner implements Runnable {

private KafkaStream<byte[], byte[]> partition;

MessageRunner(KafkaStream<byte[], byte[]> partition) {

this.partition = partition;

}

public void run() {

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

while (it.hasNext()) {

// connector.commitOffsets();手动提交offset,当autocommit.enable=false时使用

MessageAndMetadata<byte[], byte[]> item = it.next();

try{

executor.execute(new String(item.message(),charset));// UTF-8,注意异常

}catch(Exception e){

//

}

}

}

public String getContent(Message message){

ByteBuffer buffer = message.payload();

if (buffer.remaining() == 0) {

return null;

}

CharBuffer charBuffer = charset.decode(buffer);

return charBuffer.toString();

}

}

interface MessageExecutor {

public void execute(String message);

}

/\*\*

\* @param args

\*/

public static void main(String[] args) {

KafkaConsumerClient consumer = null;

try {

MessageExecutor executor = new MessageExecutor() {

public void execute(String message) {

System.out.println(message);

}

};

consumer = new KafkaConsumerClient();

consumer.setTopic("test-topic");

consumer.setPartitionsNum(2);

consumer.setExecutor(executor);

consumer.init();

Thread.sleep(10000);

} catch (Exception e) {

e.printStackTrace();

} finally {

if(consumer != null){

consumer.close();

}

}

}

}

**四、Kafka常用命令**

以下是kafka常用命令行总结：

1.查看topic的详细信息

./kafka-topics.sh -zookeeper 127.0.0.1:2181 -describe -topic testKJ1

2、为topic增加副本

./kafka-reassign-partitions.sh -zookeeper 127.0.0.1:2181 -reassignment-json-file json/partitions-to-move.json -execute

3、创建topic

./kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic testKJ1

4、为topic增加partition

./bin/kafka-topics.sh –zookeeper 127.0.0.1:2181 –alter –partitions 20 –topic testKJ1

5、kafka生产者客户端命令

./kafka-console-producer.sh --broker-list localhost:9092 --topic testKJ1

6、kafka消费者客户端命令

./kafka-console-consumer.sh -zookeeper localhost:2181 --from-beginning --topic testKJ1

7、kafka服务启动

./kafka-server-start.sh -daemon ../config/server.properties

8、下线broker

./kafka-run-class.sh kafka.admin.ShutdownBroker --zookeeper 127.0.0.1:2181 --broker #brokerId# --num.retries 3 --retry.interval.ms 60

shutdown broker

9、删除topic

./kafka-run-class.sh kafka.admin.DeleteTopicCommand --topic testKJ1 --zookeeper 127.0.0.1:2181

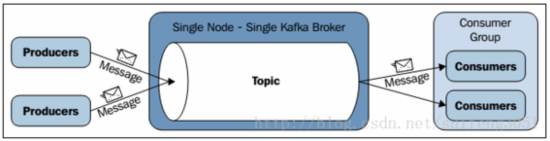
./kafka-topics.sh --zookeeper localhost:2181 --delete --topic testKJ1

10、查看consumer组内消费的offset

./kafka-run-class.sh kafka.tools.ConsumerOffsetChecker --zookeeper localhost:2181 --group test --topic testKJ1

**五、Kafka架构以及设计原理**

**1、架构图**



Producers、Kafka broker、consumers，它们分别运行在不同的节点

**2、设计思想**

**consumer group：** 各个consumer可以组成一个组，每个消息只能被组中的一个consumer消费，如果一个消息可以被多个consumer消费的话，那么这些consumer必须在不同的组。

**消息状态：** 在Kafka中，消息的状态被保存在consumer中，broker不会关心哪个消息被消费了被谁消费了，只记录一个offset值（指向partition中下一个要被消费的消息位置），这就意味着如果consumer处理不好的话，broker上的一个消息可能会被消费多次。

**消息持久化：** Kafka中会把消息持久化到本地文件系统中，并且保持极高的效率。

**消息有效期：** Kafka会长久保留其中的消息，以便consumer可以多次消费，当然其中很多细节是可配置的。

**批量发送：** Kafka支持以消息集合为单位进行批量发送，以提高push效率。

**push-and-pull:** Kafka中的Producer和consumer采用的是push-and-pull模式，即Producer只管向broker push消息，consumer只管从broker pull消息，两者对消息的生产和消费是异步的。

**Kafka集群中broker之间的关系：** 不是主从关系，各个broker在集群中地位一样，我们可以随意的增加或删除任何一个broker节点。

**负载均衡方面：** Kafka提供了一个 metadata API来管理broker之间的负载（对Kafka0.8.x而言，对于0.7.x主要靠zookeeper来实现负载均衡）。

**同步异步：** Producer采用异步push方式，极大提高Kafka系统的吞吐率（可以通过参数控制是采用同步还是异步方式）。

**分区机制partition：** Kafka的broker端支持消息分区，Producer可以决定把消息发到哪个分区，在一个分区中消息的顺序就是Producer发送消息的顺序，一个主题中可以有多个分区，具体分区的数量是可配置的。分区的意义很重大，后面的内容会逐渐体现。

**离线数据装载：** Kafka由于对可拓展的数据持久化的支持，它也非常适合向Hadoop或者数据仓库中进行数据装载

**Kafka消息的可靠性问题：**

关于消息投递可靠性，一个消息如何算投递成功，Kafka提供了三种模式

At most once—this handles the first case described. Messages are immediately marked as consumed, so they can't be given out twice, but many failure scenarios may lead to losing messages.

At least once—this is the second case where we guarantee each message will be delivered at least once, but in failure cases may be delivered twice.

Exactly once—this is what people actually want, each message is delivered once and only once.

第一种是啥都不管，发送出去就当作成功，这种情况当然不能保证消息成功投递到broker；第二种是对于Master Slave模型，只有当Master和所有Slave都接收到消息时，才算投递成功，这种模型提供了最高的投递可靠性，但是损伤了性能；第三种模型，即只要Master确认收到消息就算投递成功；实际使用时，根据应用特性选择，绝大多数情况下都会中和可靠性和性能选择第三种模型。

对于消息在broker上的可靠性，因为消息会持久化到磁盘上，所以如果正常stop一个broker，其上的数据不会丢失；但是如果不正常stop，可能会使存在页面缓存来不及写入磁盘的消息丢失，这可以通过配置flush页面缓存的周期、阈值缓解，但是同样会频繁的写磁盘会影响性能，又是一个选择题，根据实际情况配置。

接着，我们再看消息消费的可靠性，Kafka提供的是“At least once”模型，因为消息的读取进度由offset提供，offset可以由消费者自己维护也可以维护在zookeeper里，但是当消息消费后consumer挂掉，offset没有即时写回，就有可能发生重复读的情况，这种情况同样可以通过调整commit offset周期、阈值缓解，甚至消费者自己把消费和commit offset做成一个事务解决，但是如果你的应用不在乎重复消费，那就干脆不要解决，以换取最大的性能

# **六、zookeeper存储结构总图**

当我们kafka启动运行以后，就会在zookeeper上初始化kafka相关数据，主要包括六大类：

* consumers
* admin
* config
* controller
* brokers
* controller\_epoch

**1、brokers节点结构说明**

1.1 topic信息结构

/brokers/topics/[topic] :

存储某个topic的partitions所有分配信息:

Schema:

{

"version": "版本编号目前固定为数字1",

"partitions": {

"partitionId编号": [

同步副本组brokerId列表

],

"partitionId编号": [

同步副本组brokerId列表

],

.......

}

}

Example:

{

"version": 1,

"partitions": {

"0": [1, 2],

"1": [2, 1],

"2": [1, 2],

}

}

1.2 partitions信息

/brokers/topics/[topic]/partitions/[0...N] 其中[0..N]表示partition索引号

/brokers/topics/[topic]/partitions/[partitionId]/state

Schema:

{

"controller\_epoch": 表示kafka集群中的中央控制器选举次数,

"leader": 表示该partition选举leader的brokerId,

"version": 版本编号默认为1,

"leader\_epoch": 该partition leader选举次数,

"isr": [同步副本组brokerId列表]

}

Example:

{

"controller\_epoch": 1,

"leader": 2,

"version": 1,

"leader\_epoch": 0,

"isr": [2, 1]

}

1.3 broker信息

/brokers/ids/[0...N]

每个broker的配置文件中都需要指定一个数字类型的id(全局不可重复),此节点为临时znode(EPHEMERAL)

Schema:

{

"jmx\_port": jmx端口号,

"timestamp": kafka broker初始启动时的时间戳,

"host": 主机名或ip地址,

"version": 版本编号默认为1,

"port": kafka broker的服务端端口号,由server.properties中参数port确定

}

Example:

{

"jmx\_port": 5051,

"timestamp":"1403061000000"

"version": 1,

"host": "127.0.0.1",

"port": 8081

}

**2、Controller\_epoch**

/controller\_epoch -> int (epoch)

此值为一个数字,kafka集群中第一个broker第一次启动时为1，以后只要集群中center controller（中央控制器）所在broker变更或挂掉，就会重新选举新的center controller，每次center controller变更controller\_epoch值就会 + 1;

**3、Controller信息**

/controller -> int (broker id of the controller)

存储center controller（中央控制器）所在kafka broker的信息。

Schema:

{

"version": 版本编号默认为1,

"brokerid": kafka集群中broker唯一编号,

"timestamp": kafka broker中央控制器变更时的时间戳

}

Example:

{

"version": 1,

"brokerid": 3,

"timestamp": "1403061802981"

}

这个的意思就说明，当前的Controller所在的Broker机器是哪台，变更时间是多少等。

**4、Consumer信息**

/consumers/[groupId]/ids/[consumerIdString]

每个consumer都有一个唯一的ID(consumerId可以通过配置文件指定,也可以由系统生成),此id用来标记消费者信息。

Schema:

{

"version": 版本编号默认为1,

"subscription": { //订阅topic列表},

"topic名称": consumer中topic消费者线程数

"pattern": "static",

"timestamp": "consumer启动时的时间戳"

}

4.1 Consumer offset信息

/consumers/[groupId]/offsets/[topic]/[partitionId] -> long (offset)

用来跟踪每个consumer目前所消费的partition中最大的offset。此znode为持久节点，可以看出offset跟group\_id有关,以表明当消费者组(consumer group)中一个消费者失效，重新触发balance,其他consumer可以继续消费。

### **七、[为什么说Kafka使用磁盘比内存快](http://flychao88.iteye.com/blog/2273807)**

Kafka最核心的思想是使用磁盘，而不是使用内存，可能所有人都会认为，内存的速度一定比磁盘快，我也不例外。在看了Kafka的设计思想，查阅了相应资料再加上自己的测试后，发现磁盘的顺序读写速度和内存持平。

而且Linux对于磁盘的读写优化也比较多，包括read-ahead和write-behind，磁盘缓存等。如果在内存做这些操作的时候，一个是JAVA对象的内存开销很大，另一个是随着堆内存数据的增多，JAVA的GC时间会变得很长，使用磁盘操作有以下几个好处：

磁盘缓存由Linux系统维护，减少了程序员的不少工作。

磁盘顺序读写速度超过内存随机读写。

JVM的GC效率低，内存占用大。使用磁盘可以避免这一问题。

系统冷启动后，磁盘缓存依然可用。

1. **Kafka与其它Message Queue对比，以及用什么优点**

**与其它Message Queue对比对比：**

**1、RabbitMQ**

RabbitMQ是使用Erlang编写的一个开源的消息队列，本身支持很多的协议：AMQP，XMPP, SMTP, STOMP，也正因如此，它**非常重量级，更适合于企业级的开发**。同时实现了Broker构架，这意味着消息在发送给客户端时先在中心队列排队。对路由，负载均衡或者数据持久化都有很好的支持。

**2、Redis**

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

**3、ZeroMQ**

ZeroMQ号称最快的消息队列系统，尤其针对大吞吐量的需求场景。ZeroMQ能够实现RabbitMQ不擅长的高级/复杂的队列，但是开发人员需要自己组合多种技术框架，技术上的复杂度是对这MQ能够应用成功的挑战。ZeroMQ具有一个独特的非中间件的模式，你不需要安装和运行一个消息服务器或中间件，因为你的应用程序将扮演这个服务器角色。你只需要简单的引用ZeroMQ程序库，可以使用NuGet安装，然后你就可以愉快的在应用程序之间发送消息了。但是ZeroMQ仅提供非持久性的队列，也就是说如果宕机，数据将会丢失。其中，Twitter的Storm 0.9.0以前的版本中默认使用ZeroMQ作为数据流的传输（Storm从0.9版本开始同时支持ZeroMQ和Netty作为传输模块）。

**4、ActiveMQ**

ActiveMQ是Apache下的一个子项目。 类似于ZeroMQ，它能够以代理人和点对点的技术实现队列。同时类似于RabbitMQ，它少量代码就可以高效地实现高级应用场景。

**5、Kafka/Jafka**

Kafka是Apache下的一个子项目，是一个高性能跨语言分布式发布/订阅消息队列系统，而Jafka是在Kafka之上孵化而来的，即Kafka的一个升级版。具有以下特性：快速持久化，可以在O(1)的系统开销下进行消息持久化；高吞吐，在一台普通的服务器上既可以达到10W/s的吞吐速率；完全的分布式系统，Broker、Producer、Consumer都原生自动支持分布式，自动实现负载均衡；支持Hadoop数据并行加载，对于像Hadoop的一样的日志数据和离线分析系统，但又要求实时处理的限制，这是一个可行的解决方案。Kafka通过Hadoop的并行加载机制统一了在线和离线的消息处理。Apache Kafka相对于ActiveMQ是一个非常轻量级的消息系统，除了性能非常好之外，还是一个工作良好的分布式系统。

**使用Kafka消息系统优点有什么？**

**解耦**

在项目启动之初来预测将来项目会碰到什么需求，是极其困难的。消息系统在处理过程中间插入了一个隐含的、基于数据的接口层，两边的处理过程都要实现这一接口。这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。

**冗余**

有些情况下，处理数据的过程会失败。除非数据被持久化，否则将造成丢失。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。

**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。不需要改变代码、不需要调节参数。扩展就像调大电力按钮一样简单。

**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。

**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。

**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。

**缓冲**

在任何重要的系统中，都会有需要不同的处理时间的元素。例如，加载一张图片比应用过滤器花费更少的时间。消息队列通过一个缓冲层来帮助任务最高效率的执行———写入队列的处理会尽可能的快速。该缓冲有助于控制和优化数据流经过系统的速度。

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。

学习参考：

<http://www.infoq.com/cn/articles/kafka-analysis-part-1>

<http://flychao88.iteye.com/category/350737>

<http://shift-alt-ctrl.iteye.com/blog/1930791>

<http://www.tuicool.com/articles/mErEZn>