# Kafka学习文档

本教程假定您是一只小白，没有Kafka 或ZooKeeper 方面的经验。 Kafka脚本在Unix和Windows平台有所不同，在Windows平台，请使用 bin\windows\ 而不是bin/, 并将脚本扩展名改为.bat。

## Kafka概述

### 消息队列



（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。

### 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。

### 什么是Kafka

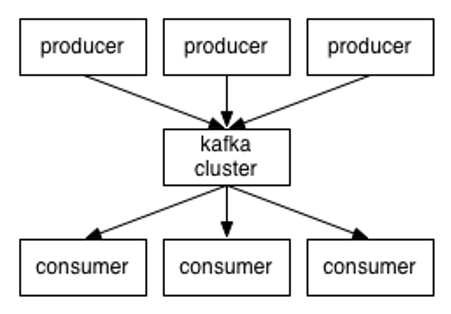
在流式计算中，Kafka一般用来缓存数据，Storm通过消费Kafka的数据进行计算。

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)称为broker。

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



Kafka架构图



Kafka详细架构图

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（不是真的复制，是概念上的）到所有的CG，但每个partion只会把消息发给该CG中的一个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。

## Kafka单节点运行方式

**Setp 1：下载代码**

下载 kafka\_2.12-2.1.0 版本并且解压。

<https://www.apache.org/dyn/closer.cgi?path=/kafka/2.1.0/kafka_2.12-2.1.0.tgz>

> tar -xzf kafka\_2.11-1.0.0.tgz

> cd kafka\_2.11-1.0.0

**Setp 2：启动服务**

Kafka 使用 ZooKeeper 如果你还没有ZooKeeper服务器，你需要先启动一个ZooKeeper服务器。 您可以通过与kafka打包在一起的便捷脚本来快速简单地创建一个单节点ZooKeeper实例。如果你有使用docker的经验，你可以使用docker-compose快速搭建一个zk集群。

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

现在启动Kafka服务器：

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

后台启动：

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

其中1>/dev/null 2>&1 是将命令产生的输入和错误都输入到空设备，也就是不输出的意思。

/dev/null代表空设备。

**Setp 3：创建一个topic**

创建一个名为“test”的topic，它有一个分区和一个副本：

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

运行list（列表）命令来查看这个topic：

> bin/kafka-topics.sh --list --zookeeper localhost:2181 test

除了手工创建topic外，你也可以配置你的broker，当发布一个不存在的topic时自动创建topic。

**Setp 4：发送消息**

Kafka自带一个命令行客户端，它从文件或标准输入中获取输入，并将其作为message（消息）发送到Kafka集群。默认情况下，每行将作为单独的message发送。

运行 producer，然后在控制台输入一些消息以发送到服务器。

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

hello world

Hello study.163.com

**Setp 5：启动消费者**

Kafka还有一个命令行使用者，它会将消息转储到标准输出。

> bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic test --from-beginning

hello world

hello study.163.com

如果在不同的终端中运行上述命令，能够在生产者终端中键入消息并看到它们出现在消费者终端中。

所有命令行工具都有选项; 运行不带参数的命令将显示使用信息。

## Kafka集群部署方式

**Setp 6：设置多 broker 集群**

到目前，我们只是单一的运行一个broker，对于Kafka，一个broker仅仅只是一个集群的大小，接下来我们来设多个broker。

首先为每个broker创建一个配置文件:

> cp config/server.properties config/server-1.properties

> cp config/server.properties config/server-2.properties

现在编辑这些新建的文件，设置以下属性：

config/server-1.properties:

    broker.id=1

    listeners=PLAINTEXT://:9093

    log.dir=/tmp/kafka-logs-1

config/server-2.properties:

    broker.id=2

    listeners=PLAINTEXT://:9094

    log.dir=/tmp/kafka-logs-2

broker.id属性是集群中每个节点的名称，这一名称是唯一且永久的。

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

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

...

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

...

现在创建一个副本为3的新topic：

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

运行命令“describe topics” 查看集群中的topic信息

> bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic my-replicated-topic

Topic:my-replicated-topic   PartitionCount:1    ReplicationFactor:3 Configs:

    Topic: my-replicated-topic  Partition: 0    Leader: 1   Replicas: 1,2,0 Isr: 1,2,0

以下是对输出信息的解释：第一行给出了所有分区的摘要，下面的每行都给出了一个分区的信息。因为我们只有一个分区，所以只有一行。

* “leader”是负责给定分区所有读写操作的节点。每个节点都是随机选择的部分分区的领导者。
* “replicas”是复制分区日志的节点列表，不管这些节点是leader还是仅仅活着。
* “isr”是一组“同步”replicas，是replicas列表的子集，它活着并被指到leader。

请注意，在示例中，节点1是该主题中唯一分区的领导者。

我们运行这个命令，看看一开始我们创建的那个test节点：

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

Topic:test  PartitionCount:1    ReplicationFactor:1 Configs:

    Topic: test Partition: 0    Leader: 0   Replicas: 0 Isr: 0

这并不奇怪，刚才创建的主题没有Replicas，并且在服务器“0”上，我们创建它的时候，集群中只有一个服务器，所以是“0”。

发布一些信息在新的topic上：

> bin/kafka-console-producer.sh --broker-list localhost:9092 --topic my-replicated-topic

...

my test message 1

my test message 2

消费这些消息：

> bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --from-beginning --topic my-replicated-topic

...

my test message 1

my test message 2

测试集群的容错，kill掉leader，Broker1作为当前的leader，也就是kill掉Broker1。

> ps aux | grep server-1.properties

7564 ttys002    0:15.91 /System/Library/Frameworks/JavaVM.framework/Versions/1.8/Home/bin/java...

> kill -9 7564

备份节点之一成为新的leader，而broker1已经不在同步备份集合里了。

> bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic my-replicated-topic

Topic:my-replicated-topic   PartitionCount:1    ReplicationFactor:3 Configs:

    Topic: my-replicated-topic  Partition: 0    Leader: 2   Replicas: 1,2,0 Isr: 2,0

即使最初接受写入的领导者已经失败，这些消息仍可供消费：

> bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --from-beginning --topic my-replicated-topic

...

my test message 1

my test message 2

**Setp 7：使用 Kafka Connect 导入/导出数据**

Kafka Connect是Kafka的一个工具，它可以将数据导入和导出到Kafka。它是一种可扩展工具，通过运行*connectors（连接器）*， 使用自定义逻辑来实现与外部系统的交互。接下来我们将学习如何使用简单的connectors来运行Kafka Connect，这些connectors 将文件中的数据导入到Kafka topic中，并从中导出数据到一个文件。

首先，我们将创建一些种子数据来进行测试：

> echo -e "allen" > test.txt

> echo -e "tony" >> test.txt

接下来，我们将启动两个*standalone（独立）*运行的连接器，第一个是源连接器，它从输入文件读取行并生成Kafka主题，第二个是宿连接器从Kafka主题读取消息并将每个消息生成为输出文件中的一行。

> bin/connect-standalone.sh config/connect-standalone.properties config/connect-file-source.properties config/connect-file-sink.properties

一旦Kafka Connect进程启动，源连接器应该开始从test.txt主题读取行并生成它们connect-test，并且接收器连接器应该开始从主题读取消息connect-test 并将它们写入文件test.sink.txt。我们可以通过检查输出文件的内容来验证数据是否已通过整个管道传递：

> more test.sink.txt

allen

tony

数据存储在Kafka主题中connect-test，因此我们还可以运行控制台使用者来查看主题中的数据：

> bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic connect-test --from-beginning

{"schema":{"type":"string","optional":false},"payload":"allen"}

{"schema":{"type":"string","optional":false},"payload":"tony"}

...

连接器一直在处理数据，所以我们可以将数据添加到文件中，并看到它在pipeline 中移动：

> echo mike >> test.txt

## Kafka Stream

Kafka Streams是一个客户端库，用于构建任务关键型实时应用程序和微服务，其中输入和/或输出数据存储在Kafka集群中。Kafka Streams结合了在客户端编写和部署标准Java和Scala应用程序的简单性以及Kafka服务器端集群技术的优势，使这些应用程序具有高度可扩展性，弹性，容错性，分布式等等。

以下是[WordCountDemo](https://github.com/apache/kafka/blob/2.1/streams/examples/src/main/java/org/apache/kafka/streams/examples/wordcount/WordCountDemo.java)示例代码的要点（为了方便阅读，使用的是java8 lambda表达式）。

步骤：

1.启动zk和kafka

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

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

1. 准备输入主题并启动生产者

创建名为**streams-plaintext-input**的输入主题和名为**streams-wordcount-output**的输出主题：

> bin/kafka-topics.sh --create \

    --zookeeper localhost:2181 \

    --replication-factor 1 \

    --partitions 1 \

    --topic streams-plaintext-input

Created topic "streams-plaintext-input".

注意：我们创建输出主题并启用压缩，因为输出流是更改日志流

> bin/kafka-topics.sh --create \

    --zookeeper localhost:2181 \

    --replication-factor 1 \

    --partitions 1 \

    --topic streams-wordcount-output \

    --config cleanup.policy=compact

Created topic "streams-wordcount-output".

使用相同的**kafka-topics**工具描述创建的主题：

> bin/kafka-topics.sh --zookeeper localhost:2181 --describe

1. 启动Wordcount应用程序

> bin/kafka-run-class.sh org.apache.kafka.streams.examples.wordcount.WordCountDemo

演示应用程序将从输入主题**stream-plaintext-input**读取，对每个读取消息执行WordCount算法的计算，并将其当前结果连续写入输出主题**streams-wordcount-output**。

1. 处理数据

开启一个生产者终端：

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

streams-plaintext-input

all streams lead to kafka

开启一个消费者终端：

> bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 \

    --topic streams-wordcount-output \

    --from-beginning \

    --formatter kafka.tools.DefaultMessageFormatter \

    --property print.key=true \

    --property print.value=true \

    --property key.deserializer=org.apache.kafka.common.serialization.StringDeserializer \

--property value.deserializer=org.apache.kafka.common.serialization.LongDeserializer

all     1

streams 1

lead    1

to      1

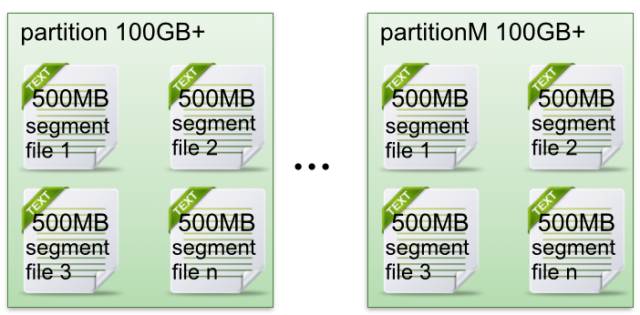
kafka   1

这里，第一列是java.lang.String格式的Kafka消息键，表示正在计数的单词，第二列是java.lang.Long格式的消息值，表示单词的最新计数。

## Kafka分片存储机制

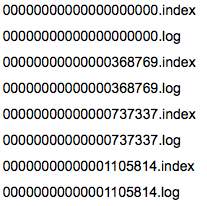
* 复习几个kafka重要概念：
* **Broker**：消息中间件处理结点，一个Kafka节点就是一个broker，多个broker可以组成一个Kafka集群。
* **Topic**：一类消息，例如page view日志、click日志等都可以以topic的形式存在，Kafka集群能够同时负责多个topic的分发。
* **Partition**：topic物理上的分组，一个topic可以分为多个partition，每个partition是一个有序的队列。
* **Segment**：partition物理上由多个segment组成，下面有详细说明。
* **offset**：每个partition都由一系列有序的、不可变的消息组成，这些消息被连续的追加到partition中。partition中的每个消息都有一个连续的序列号叫做offset,用于partition中唯一标识的这条消息。
* topic中partition存储分布

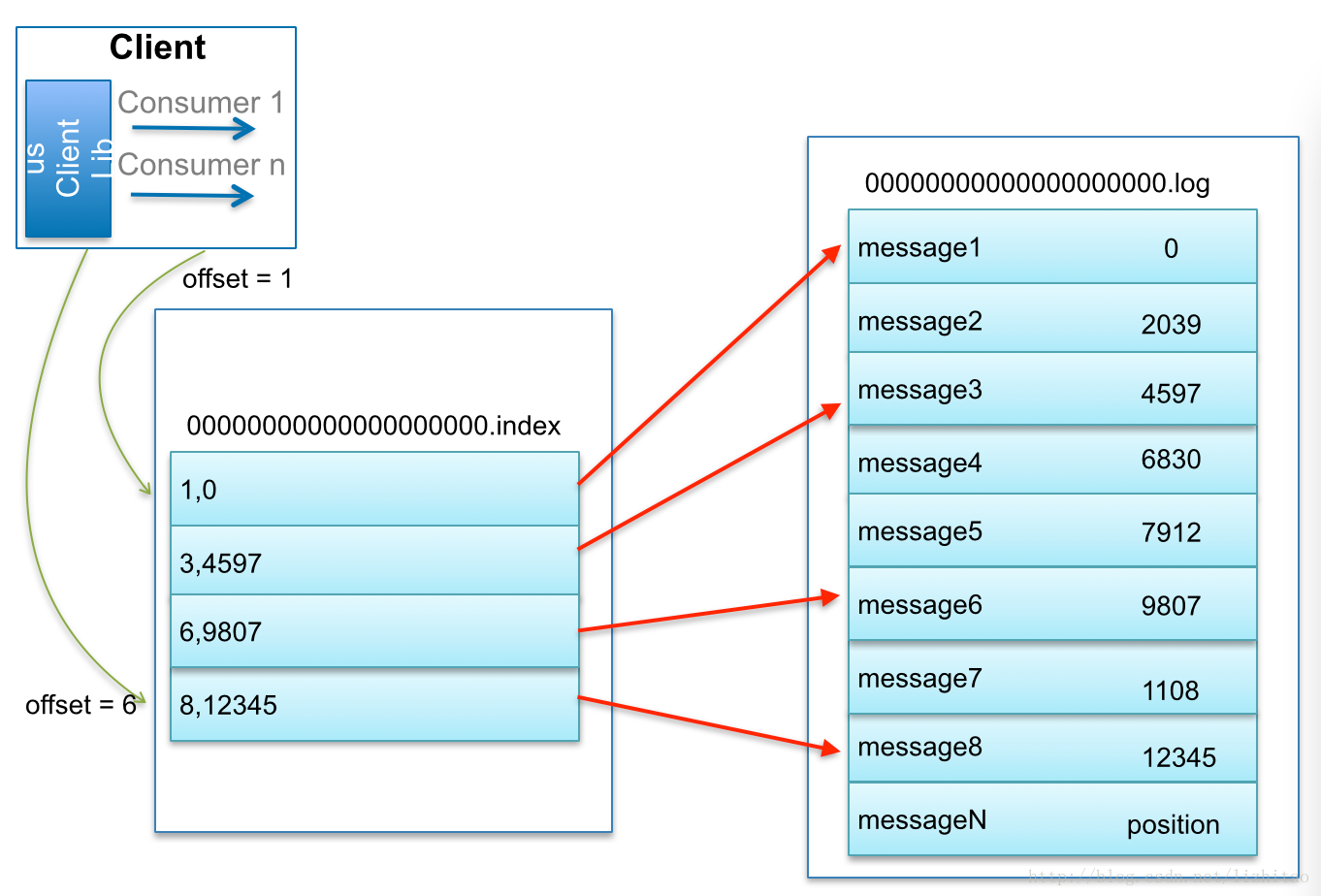
下面示意图形象说明了partition中文件存储方式:



* 每个partion(目录)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。（默认情况下每个文件大小为1G）
* 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。

这样做的好处就是能快速删除无用文件，有效提高磁盘利用率。





## Kafka消息分发和消费者push、pull机制

### producer消息分发

* **Producer客户端负责消息的分发**
* kafka集群中的任何一个broker都可以向producer提供metadata信息,这些metadata中包含”集群中存活的servers列表”/”partitions leader列表”等信息；
* 当producer获取到metadata信息之后, producer将会和Topic下所有partition leader保持socket连接；
* 消息由producer直接通过socket发送到broker，中间不会经过任何”路由层”，事实上，消息被路由到哪个partition上由producer客户端决定；比如可以采用”random”“key-hash”“轮询”等,如果一个topic中有多个partitions，那么在producer端实现”消息均衡分发”是必要的。
* 在producer端的配置文件中,开发者可以指定partition路由的方式。
* **Producer消息发送的应答机制**

设置发送数据是否需要服务端的反馈,有三个值0,1,all

* 0: producer不会等待broker发送ack
* 1: 当leader接收到消息之后发送ack
* all: 当所有的follower都同步消息成功后发送ack

默认值为：request.required.acks=1

### consumer push、pull机制

consumer以 poll（拉取）的方式从broker获取消息。

consumer获取消息，可以使用两种方式：push或poll模式。下面我们简单介绍一下这两种区别：

* **push模式**

常见的push模式如storm的消息处理，由spout负责消息的推送。该模式下需要一个中心节点，负责消息的分配情况（哪段消息分配给consumer1，哪段消息分配给consumer2），同时还要监听consumer的ack消息用于判断消息是否处理成功，如果在timeout时间内未收到响应可以认为该consumer挂掉，需要重新分配sonsumer上失败的消息。这种模式有个问题，不太容易实现我们想要的消息回放功能，因为理想情况下由consumer决定我到底要消费什么，而这种模式完全由master决定。

* **pull模式**

如上图模式，该模式为pull模式，由consumer决定消息的消费情况，这种模式有一个好处是我们不需要返回ack消息，因为当consumer申请消费下一批消息时就可以认为上一批消息已经处理完毕，也不需要处理超时的问题，consumer可以根据自己的消费能力来消费消息。但这个还有一个问题，如何保证处理的消息的不会重复呢，kafka具体做法就是增加队列的并发度（partition），可以一个partition对准一个consumer。

综上，kafka的consumer之所以没有采用push模式，是因为push模式很难适应消费者速率不同的消费者而且很难实现消息的回放功能，因为消息发送速率是由broker决定的。push模式的目标就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message。

### 6.3 pull与push的区别

pull技术：

客户机向服务器请求信息；

kafka中，consuemr根据自己的消费能力以适当的速率消费信息；

push技术：

服务器主动将信息发往客户端的技术；

push模式的目标就是尽可能以最快的速率传递消息。

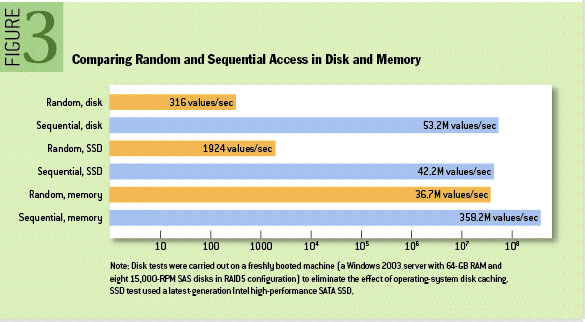
## Kafka持久化

### 概述

**不要畏惧文件系统!**

Kafka大量依赖文件系统去存储和缓存消息。对于硬盘有个传统的观念是硬盘总是很慢，这使很多人怀疑基于文件系统的架构能否提供优异的性能。实际上硬盘的快慢完全取决于使用它的方式。设计良好的硬盘架构可以和内存一样快。

在6块7200转的SATA RAID-5磁盘阵列的线性写速度差不多是600MB/s，但是随机写的速度却是100k/s，差了差不多6000倍。现在的操作系统提供了预读取和后写入的技术。实际上线性的访问磁盘，很多时候比随机的内存访问快得多。



为了提高性能，现代操作系统往往使用内存作为磁盘的缓存，现代操作系统乐于把所有空闲内存用作磁盘缓存，虽然这可能在缓存回收和重新分配时牺牲一些性能。所有的磁盘读写操作都会经过这个缓存，这不太可能被绕开除非直接使用I/O。所以虽然每个程序都在自己的线程里只缓存了一份数据，但在操作系统的缓存里还有一份，这等于存了两份数据。

* 基于jvm内存有以下缺点：
* Java对象占用空间是非常大的，差不多是要存储的数据的两倍甚至更高。
* 随着堆中数据量的增加，垃圾回收回变的越来越困难，而且可能导致错误

基于以上分析，如果把数据缓存在内存里，因为需要存储两份，不得不使用两倍的内存空间，Kafka基于JVM，又不得不将空间再次加倍,再加上要避免GC带来的性能影响，在一个32G内存的机器上，不得不使用到28-30G的内存空间。并且当系统重启的时候，又必须要将数据刷到内存中（ 10GB 内存差不多要用10分钟），就算使用冷刷新（不是一次性刷进内存，而是在使用数据的时候没有就刷到内存）也会导致最初的时候新能非常慢。

* 基于操作系统的文件系统来设计有以下好处：
* 可以通过os的pagecache来有效利用主内存空间，由于数据紧凑，可以cache大量数据，并且没有gc的压力
* 即使服务重启，缓存中的数据也是热的（不需要预热）。而基于进程的缓存，需要程序进行预热，而且会消耗很长的时间。（10G大概需要10分钟）
* 大大简化了代码。因为在缓存和文件系统之间保持一致性的所有逻辑都在OS中。以上建议和设计使得代码实现起来十分简单，不需要尽力想办法去维护内存中的数据，数据会立即写入磁盘。

总的来说，Kafka不会保持尽可能多的内容在内存空间，而是尽可能把内容直接写入到磁盘。所有的数据都及时的以持久化日志的方式写入到文件系统，而不必要把内存中的内容刷新到磁盘中。

### 日志数据持久化特性

写操作：通过将数据追加到文件中实现

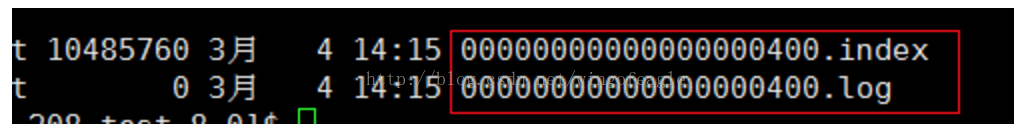
读操作：读的时候从文件中读就好了

### 优势

* 读操作不会阻塞写操作和其他操作（因为读和写都是追加的形式，都是顺序的，不会乱，所以不会发生阻塞），数据大小不对性能产生影响；
* 没有容量限制（相对于内存来说）的硬盘空间建立消息系统；
* 线性访问磁盘，速度快，可以保存任意一段时间！

### 持久化原理

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的吞吐率可以水平扩展，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**。



每个日志文件都是“log entries”序列，每一个log entry包含一个4字节整型数（值为N），其后跟N个字节的消息体。每条消息都有一个当前partition下唯一的64字节的offset，它指明了这条消息的起始位置。磁盘上存储的消息格式如下：

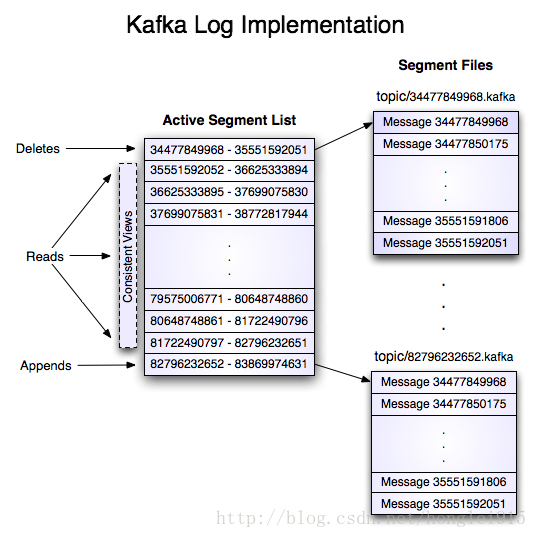
**消息长度: 4 bytes (value: 1 + 4 + n)**

**版本号: 1 byte**

**CRC校验码: 4 bytes**

**具体的消息: n bytes**

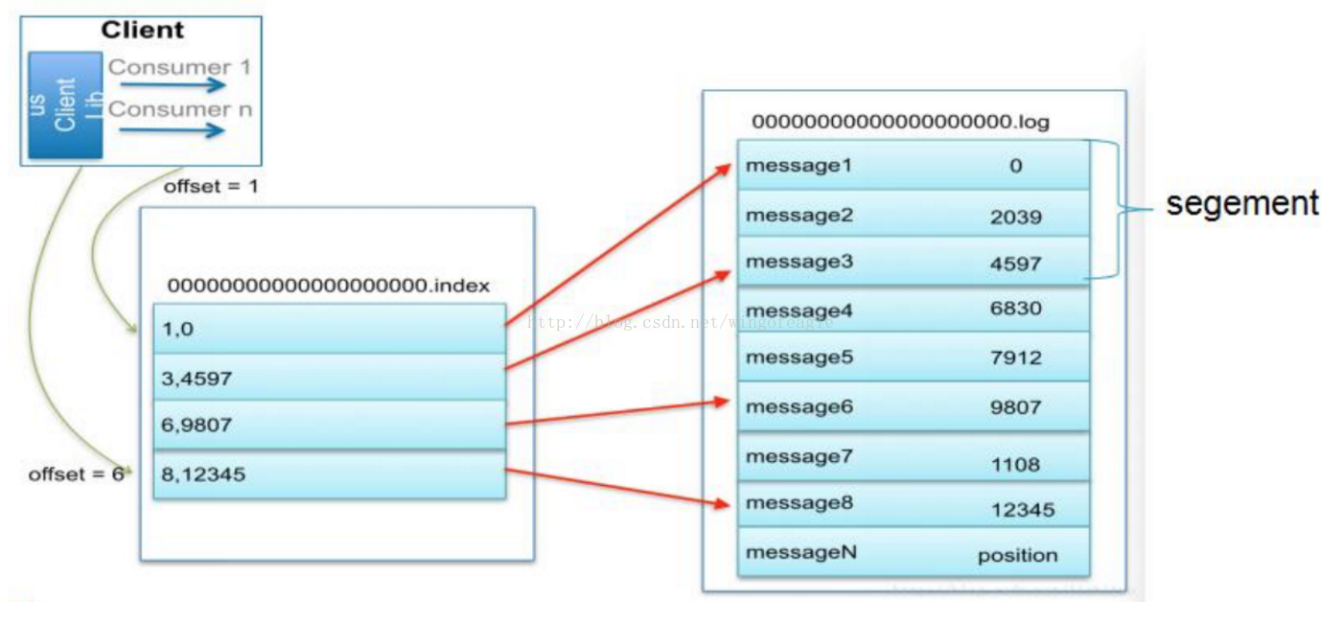
这个“log entries”并非由一个文件构成，而是分成多个segment，每个segment名为该segment第一条消息的offset和“.kafka”组成。另外会有一个索引文件，它标明了每个segment下包含的log entry的offset范围，如下图所示：



### 索引

稀疏存储，每隔一定字节的数据建立一条索引（这样的目的是为了减少索引文件的大小）。

下图为一个partition的索引示意图：

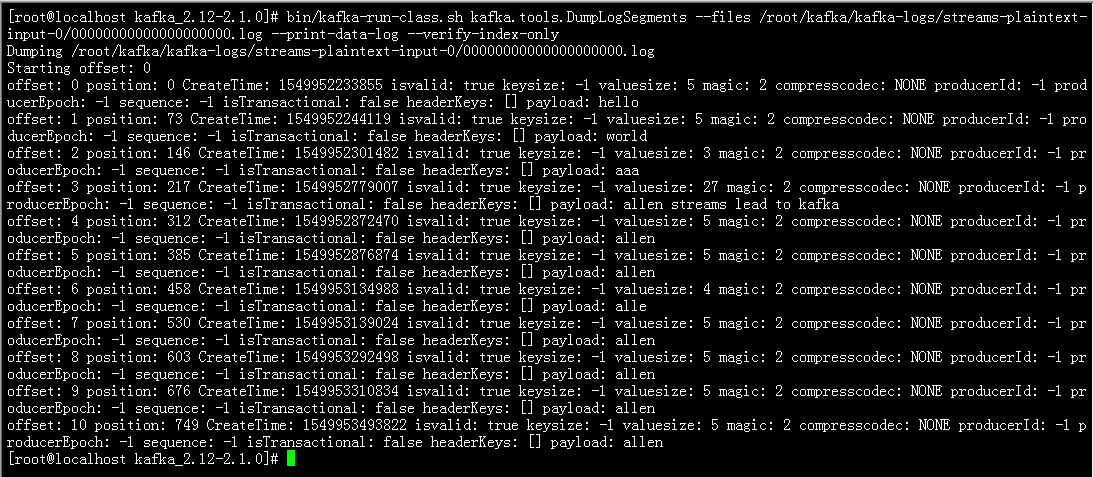


**注**：

1. 现在对6.和8建立了索引，如果要查找7，则会先查找到8然后，再找到8后的一个索引6，然后两个索引之间做二分法，找到7的位置
2. 每一个log文件中又分为多个segment

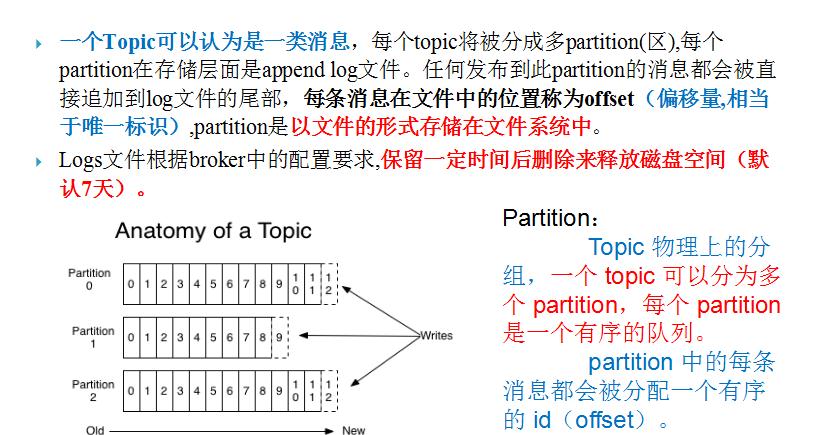
通过调用kafka自带的工具，可以看到日志下的数据信息

> bin/kafka-run-class.sh kafka.tools.DumpLogSegments --files /root/kafka/kafka-logs/streams-plaintext-input-0/00000000000000000000.log --print-data-log --verify-index-only



kafka日志分为index与log，两个成对出现；index文件存储元数据（用来描述数据的数据，这也可能是为什么index文件这么大的原因了），log存储消息。索引文件元数据指向对应log文件中message的迁移地址；例如2,128指log文件的第2条数据，偏移地址为128；而物理地址（在index文件中指定）+ 偏移地址可以定位到消息。

因为每条消息都被append到该partition中，是顺序写磁盘，因此效率非常高（经验证，顺序写磁盘效率比随机写内存还要高，这是Kafka高吞吐率的一个很重要的保证）。



## Leader选举机制

### Kafka的Leader是什么

首先Kafka会将接收到的消息分区（partition），每个主题（topic）的消息有不同的分区。这样一方面消息的存储就不会受到单一服务器存储空间大小的限制，另一方面消息的处理也可以在多个服务器上并行。

其次为了保证高可用，每个分区都会有一定数量的副本（replica）。这样如果有部分服务器不可用，副本所在的服务器就会接替上来，保证应用的持续性。

但是，为了保证较高的处理效率，消息的读写都是在固定的一个副本上完成。这个副本就是所谓的Leader，而其他副本则是Follower。而Follower则会定期地到Leader上同步数据。

### Leader选举

  如果某个分区所在的服务器出了问题，不可用，kafka会从该分区的其他的副本中选择一个作为新的Leader。之后所有的读写就会转移到这个新的Leader上。现在的问题是应当选择哪个作为新的Leader。显然，只有那些跟Leader保持同步的Follower才应该被选作新的Leader。

  Kafka会在Zookeeper上针对每个Topic维护一个称为ISR（in-sync replica，已同步的副本）的集合，该集合中是一些分区的副本。只有当这些副本都跟Leader中的副本同步了之后，kafka才会认为消息已提交，并反馈给消息的生产者。如果这个集合有增减，kafka会更新zookeeper上的记录。

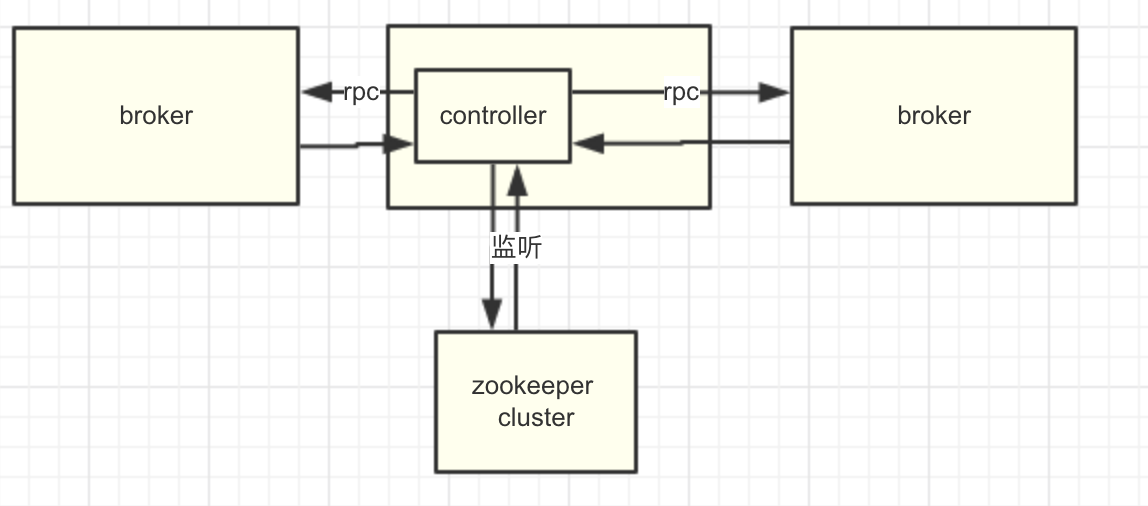
  如果某个分区的Leader不可用，Kafka就会从ISR集合中选择一个副本作为新的Leader。

  显然通过ISR，kafka需要的冗余度较低，可以容忍的失败数比较高。假设某个topic有f+1个副本，kafka可以容忍f个服务器不可用。

### 具体选举过程

最简单最直观的方案是，leader在zk上创建一个临时节点，所有Follower对此节点注册监听，当leader宕机时，此时ISR里的所有Follower都尝试创建该节点，而创建成功者（Zookeeper保证只有一个能创建成功）即是新的Leader，其它Replica即为Follower。

实际上的实现思路也是这样，只是优化了下，多了个代理控制管理类（controller）。**引入的原因是，当kafka集群业务很多，partition达到成千上万时，当broker宕机时，造成集群内大量的调整，会造成大量Watch事件被触发，Zookeeper负载会过重。**zk是不适合大量写操作的。



contoller，zk，其他broker交互流程图

* Controller提供：
* 增加删除topic
* 更新分区副本数量
* 选举分区leader
* 集群broker增加和宕机后的调整
* 自身的选举controller leader功能

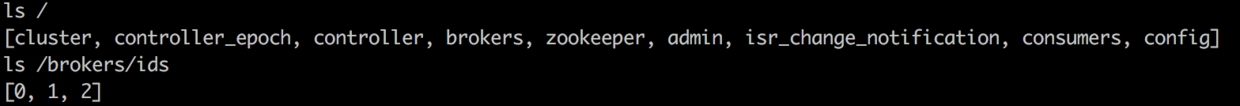
这些功能都是controller通过监听Zookeeper间接节点出发，然后controller再跟其他的broker具体的去交互实现的（rpc的方式）。

* controller的内部设计：

当前controller启动时会为集群中所有broker创建一个各自的连接。假设你的集群中有100台broker，那么controller启动时会创建100个Socket连接(也包括与它自己的连接！)。具体的类NetworkClient类，底层就是Java NIO reactor模型)。Controller会为每个连接都创建一个对应的请求发送线程（RequestSendThread）。

controller实现如上功能，要先熟悉kafka下zk上的数据存储结构：

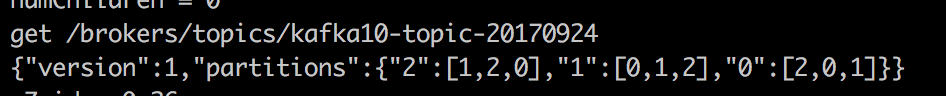
* brokers列表：ls /brokers/ids
* 某个broker信息：get /brokers/ids/0
* topic信息：get /brokers/topics/kafka10-topic-xxx
* partition信息：get /brokers/topics/kafka10-topic-xxx/partitions/0/state
* controller中心节点变更次数：get /controller\_epoch
* conrtoller leader信息：get /controller



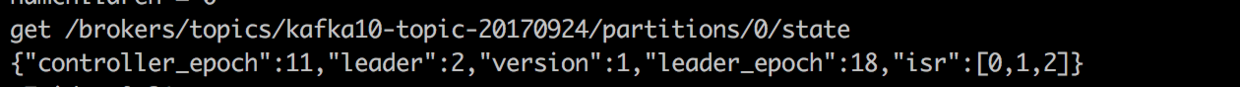
broker机器id

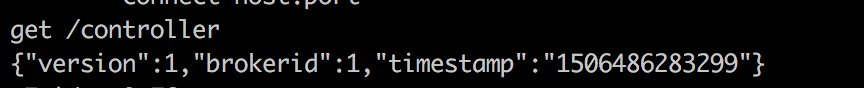
IMG_256

某个broker信息



topic信息

partition信息

conrtoller leader信息

### 为什么不用少数服用多数的方法

少数服从多数是一种比较常见的一致性算法和Leader选举法。它的含义是只有超过半数的副本同步了，系统才会认为数据已同步；选择Leader时也是从超过半数的同步的副本中选择。这种算法需要较高的冗余度。譬如只允许一台机器失败，需要有三个副本；而如果只容忍两台机器失败，则需要五个副本。而kafka的ISR集合方法，分别只需要两个和三个副本。

如果所有的ISR副本都失败了怎么办

  此时有两种方法可选，一种是等待ISR集合中的副本复活，一种是选择任何一个立即可用的副本，而这个副本不一定是在ISR集合中。这两种方法各有利弊，实际生产中按需选择。

  如果要等待ISR副本复活，虽然可以保证一致性，但可能需要很长时间。而如果选择立即可用的副本，则很可能该副本并不一致。

## Kafka API实战

低级及高级api使用

## [Kafka producer拦截器(interceptor)](http://www.cnblogs.com/huxi2b/p/7072447.html)

### 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。

### 拦截器案例

1）需求：

实现一个简单的双interceptor组成的拦截链。第一个interceptor会在消息发送前将时间戳信息加到消息value的最前部；第二个interceptor会在消息发送后更新成功发送消息数或失败发送消息数。



2）案例实操

（1）增加时间戳拦截器

import java.util.Map;

import org.apache.kafka.clients.producer.ProducerInterceptor;

import org.apache.kafka.clients.producer.ProducerRecord;

import org.apache.kafka.clients.producer.RecordMetadata;

public class TimeInterceptor implements ProducerInterceptor<String, String> {

@Override

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

}

@Override

public ProducerRecord<String, String> onSend(ProducerRecord<String, String> record) {

// 创建一个新的record，把时间戳写入消息体的最前部

return new ProducerRecord(record.topic(), record.partition(), record.timestamp(), record.key(),

System.currentTimeMillis() + "," + record.value().toString());

}

@Override

public void onAcknowledgement(RecordMetadata metadata, Exception exception) {

}

@Override

public void close() {

}

}

（2）统计发送消息成功和发送失败消息数，并在producer关闭时打印这两个计数器

import java.util.Map;

import org.apache.kafka.clients.producer.ProducerInterceptor;

import org.apache.kafka.clients.producer.ProducerRecord;

import org.apache.kafka.clients.producer.RecordMetadata;

public class CounterInterceptor implements ProducerInterceptor<String, String>{

private int errorCounter = 0;

private int successCounter = 0;

@Override

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

}

@Override

public ProducerRecord<String, String> onSend(ProducerRecord<String, String> record) {

return record;

}

@Override

public void onAcknowledgement(RecordMetadata metadata, Exception exception) {

// 统计成功和失败的次数

if (exception == null) {

successCounter++;

} else {

errorCounter++;

}

}

@Override

public void close() {

// 保存结果

System.out.println("Successful sent: " + successCounter);

System.out.println("Failed sent: " + errorCounter);

}

}

（3）producer主程序

import java.util.ArrayList;

import java.util.List;

import java.util.Properties;

import org.apache.kafka.clients.producer.KafkaProducer;

import org.apache.kafka.clients.producer.Producer;

import org.apache.kafka.clients.producer.ProducerConfig;

import org.apache.kafka.clients.producer.ProducerRecord;

public class InterceptorProducer {

public static void main(String[] args) throws Exception {

// 1 设置配置信息

Properties props = new Properties();

props.put("bootstrap.servers", "localhost:9092");

props.put("acks", "all");

props.put("retries", 0);

props.put("batch.size", 16384);

props.put("linger.ms", 1);

props.put("buffer.memory", 33554432);

props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");

props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

// 2 构建拦截链

List<String> interceptors = new ArrayList<>();

interceptors.add("com.atguigu.kafka.interceptor.TimeInterceptor"); interceptors.add("com.atguigu.kafka.interceptor.CounterInterceptor");

props.put(ProducerConfig.INTERCEPTOR\_CLASSES\_CONFIG, interceptors);

String topic = "first";

Producer<String, String> producer = new KafkaProducer<>(props);

// 3 发送消息

for (int i = 0; i < 10; i++) {

ProducerRecord<String, String> record = new ProducerRecord<>(topic, "message" + i);

producer.send(record);

}

// 4 一定要关闭producer，这样才会调用interceptor的close方法

producer.close();

}

}

3）测试

（1）在kafka上启动消费者，然后运行客户端java程序。

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

1501904047034,message0

1501904047225,message1

1501904047230,message2

1501904047234,message3

1501904047236,message4

1501904047240,message5

1501904047243,message6

1501904047246,message7

1501904047249,message8

1501904047252,message9

（2）观察java平台控制台输出数据如下：

Successful sent: 10

Failed sent: 0

## Kafka 自定义分区器

在调用Kafka的Producer API时，如果没有指定分区器，那么数据将会根据默认分区器的算法均分到各个分区。然而实际的生产环境中，可能Kafka的分区数不止一个(官方建议：Kafka的分区数量应该是Broker数量的整数倍！)，所以这时需要我们自定义分区器。

### 默认分区器DefaultPartitioner

默认分区器：

org.apache.kafka.clients.producer.internals.DefaultPartitioner

默认分区器获取分区：

如果消息的 key 为 null，此时 producer 会使用默认的 partitioner 分区器将消息随机分布到 topic 的可用 partition 中。

如果 key 不为 null，并且使用了默认的分区器，kafka 会使用自己的 hash 算法对 key 取 hash 值，使用 hash 值与 partition 数量取模，从而确定发送到哪个分区。注意：此时 key 相同的消息会发送到相同的分区(只要 partition 的数量不变化)。

/\*\*

\* Compute the partition for the given record.

\* 计算partition

\* @param topic The topic name

\* @param key The key to partition on (or null if no key)

\* @param keyBytes serialized key to partition on (or null if no key)

\* @param value The value to partition on or null

\* @param valueBytes 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) {

//获取指定topic的partitions

List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);

int numPartitions = partitions.size();

//key=null

if (keyBytes == null) {

int nextValue = nextValue(topic);

//可用分区

List<PartitionInfo> availablePartitions = cluster.availablePartitionsForTopic(topic);

if (availablePartitions.size() > 0) {

//消息随机分布到topic可用的partition中

int part = Utils.toPositive(nextValue) % availablePartitions.size();

return availablePartitions.get(part).partition();

} else {

// no partitions are available, give a non-available partition

return Utils.toPositive(nextValue) % numPartitions;

}

//如果 key 不为 null，并且使用了默认的分区器，kafka 会使用自己的 hash 算法对 key 取 hash 值

} else {//通过hash获取partition

// hash the keyBytes to choose a partition

return Utils.toPositive(Utils.murmur2(keyBytes)) % numPartitions;

}

}

### 自定义分区器

查看源码可以发现：

1、DefaultPartitioner实现了Partitioner接口

2、分区算法的实现在这个方法中：

public int partition(String topic, Object key, byte[] keyBytes, Object value, byte[] valueBytes, Cluster cluster){…………}

3、如果我们需要实现自己的分区器，那么可以有2种方法

(1)新建一个包路径和DefaultPartitioner所在的路径一致，然后更改

public int partition(String topic, Object key, byte[] keyBytes, Object value, byte[] valueBytes, Cluster cluster){…………}

方法体的内容，更改为我们自己的算法即可。

(2)新建一个类，实现Partitioner接口

public class MySamplePartitioner implements Partitioner {

private final AtomicInteger counter = new AtomicInteger(new Random().nextInt());

private Random random = new Random();

//我的分区器定义

@Override

public int partition(String topic, Object key, byte[] keyBytes, Object value, byte[] valueBytes, Cluster cluster) {

List<PartitionInfo> partitioners = cluster.partitionsForTopic(topic);

int numPartitions = partitioners.size();

/\*\*

\* 由于我们按key分区，在这里我们规定：key值不允许为null。

\* 在实际项目中，key为null的消息\*，可以发送到同一个分区,或者随机分区。

\*/

int res = 1;

if (keyBytes == null) {

System.out.println("value is null");

res = random.nextInt(numPartitions);

} else {

// System.out.println("value is " + value + "\n hashcode is " + value.hashCode());

res = Math.abs(key.hashCode()) % numPartitions;

}

System.out.println("data partitions is " + res);

return res;

}

}

### 分区器使用

1、如果重载默认分区器，那么不用在Producer中做修改

2、如果是实现Partitioner接口的方式，那么需要在Producer中添加一个属性：

props.put("partitioner.class","com.study.kafka.MySamplePartitioner");//我的自定义分区器

partitioner.class值为自定义分区类的完整包名，这样生产者就会选择自定义的分区策略。



说明：

1.客户端测试环境中，自定义分区类跟生产者类在一个项目中，不需要其他操作；

2.想要自定义的分区放到kafka的服务器端环境时，需要将自定义的分区类生成jar包放到kafka环境的lib下，同样配置文件中指定完整包名。

当然我们还可以在不使用默认分区和自定义分区器的情况下直接指定分区直，具体做法：

在创建ProducerRecord时，指定消息的分区值。

int partition = 0;

if(key<100){

partition = 0;

}else if(key<200){

partition = 1;

}else{

partition = 2;

}

ProducerRecord<String,String> records = new ProducerRecord<String,String>(TOPIC,partition,key,value);

kafkaProducer.send(records);

## Kafka扩容

扩容：增加机器，例如原来三台服务器的kafka集群增加两台机器成为有五台机器的kafka集群，跟搭建差不多

分区重新分配：在原来机器上的主题分区不会自动均衡到新的机器，需要使用分区重新分配工具来均衡均衡

重新分配官方文档地址：<http://kafka.apache.org/documentation/#basic_ops_cluster_expansion>

将服务器添加到Kafka集群很简单，只需为它们分配一个唯一的代理ID，并在新服务器上启动Kafka。但是，这些新服务器不会自动分配任何数据分区，因此除非将分区移动到它们，否则在创建新主题之前它们不会执行任何工作。因此，通常在将计算机添加到群集时，您需要将一些现有数据迁移到这些计算机。

迁移数据的过程是手动启动的，但完全自动化。在幕后，Kafka将添加新服务器作为其正在迁移的分区的跟随者，并允许它完全复制该分区中的现有数据。当新服务器完全复制此分区的内容并加入同步副本时，其中一个现有副本将删除其分区的数据。

分区重新分配工具可用于在代理之间移动分区。理想的分区分布将确保所有代理的均匀数据负载和分区大小。分区重新分配工具无法自动研究Kafka群集中的数据分布并移动分区以实现均匀的负载分配。因此，管理员必须弄清楚应该移动哪些主题或分区。

### 自动将数据迁移到新计算机

分区重新分配工具可用于将一些主题从当前的代理集移动到新添加的代理。这在扩展现有集群时通常很有用，因为将整个主题移动到新的代理集更容易，而不是一次移动一个分区。当用于执行此操作时，用户应提供应移至新的代理集的主题列表和新代理的目标列表。然后，该工具在新的代理集中均匀分配给定主题列表的所有分区。在此移动期间，主题的复制因子保持不变。有效地，输入主题列表的所有分区的副本将从旧的代理集移动到新添加的代理。

例如，以下示例将主题foo1的所有分区移动到新的代理集5,6。在此移动结束时，主题foo1的所有分区将仅存在于代理5,6上。

由于该工具接受主题的输入列表作为json文件，因此首先需要确定要移动的主题并创建json文件，如下所示：

{"topics": [{"topic": "foo1"}],

"version":1

}

一旦json文件准备就绪，使用分区重新分配工具生成候选分配：

> bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --topics-to-move-json-file topics-to-move.json --broker-list "3,4" --generate

Current partition replica assignment

{"version":1,

"partitions":[{"topic":"foo1","partition":1,"replicas":[1,2]},

{"topic":"foo1","partition":0,"replicas":[1,2]}]

}

Proposed partition reassignment configuration

{"version":1,

"partitions":[{"topic":"foo1","partition":1,"replicas":[5,6]},

{"topic":"foo1","partition":0,"replicas":[5,6]}]

}

该工具生成一个候选分配，将所有分区从主题foo1移动到代理5,6。但请注意，此时分区移动尚未开始，它只是告诉您当前的分配和建议的新分配。应保存当前分配，以防您想要回滚它。新的赋值应保存在json文件中（例如expand-cluster-reassignment.json），以使用--execute选项输入到工具，如下所示：

> bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --reassignment-json-file expand-cluster-reassignment.json --execute

最后，--verify选项可与该工具一起使用，以检查分区重新分配的状态。请注意，相同的expand-cluster-reassignment.json（与--execute选项一起使用）应与--verify选项一起使用：

> bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --reassignment-json-file expand-cluster-reassignment.json --verify

Status of partition reassignment:

Reassignment of partition [foo1,0] completed successfully

Reassignment of partition [foo1,1] is in progress

Reassignment of partition [foo1,2] is in progress

Reassignment of partition [foo2,0] completed successfully

Reassignment of partition [foo2,1] completed successfully

Reassignment of partition [foo2,2] completed successfully

### 减少迁移的数据量

如果要迁移的Topic 有大量数据（假如Topic 默认保留1天的数据），可以在迁移之前临时动态地调整retention.ms 来减少数据量，Kafka 会主动purge 掉1个小时之前的数据。

> bin/kafka-topics --zookeeper localhost:2181 --alter --topic sdk\_counters --config retention.ms=3600000

在迁移完成后，恢复原先设置

> bin/kafka-topics --zookeeper 10.1.1.50:2181/kafka --alter --topic sdk\_counters --config retention.ms=86400000

### 重新制定partition leader

有时候由于节点down 了，partition 的leader 可能不是我们希望的那个的，这时，可以通过kafka-preferred-replica-election 工具将replica 中的第一个节点作为该分区的leader。

手动编辑topicPartitionList.json 文件，指定要重新分配leader 的分区。

{"partitions":[{"topic":"sdk\_counters","partition":5}]}

执行命令

> bin/kafka-preferred-replica-election --zookeeper 10.1.1.50:2181/kafka -path-to-json-file ~/kafka/topicPartitionList.json

### 中断迁移任务

重新指定partition leader一旦启动reassign 脚本，则无法停止迁移任务。如果需要强制停止，可以通过zookeeper 进行修改。

> bin/zkCli.sh -server localhost:2181

> delete /admin/reassign\_partitions

### 自定义分区分配和迁移

分区重新分配工具还可用于选择性地将分区的副本移动到特定的代理集。当以这种方式使用时，假设用户知道重新分配计划并且不需要工具生成候选重新​​分配，有效地跳过 - 生成步骤并直接移动到--execute步骤

例如，以下示例将主题foo1的分区0移动到代理5,6，将主题foo2的分区1移动到代理2,3：

第一步是在json文件中手工制作自定义重新分配计划：

> cat custom-reassignment.json

{"version":1,"partitions":[{"topic":"foo1","partition":0,"replicas":[5,6]},{"topic":"foo2","partition":1,"replicas":[2,3]}]}

然后，使用带有--execute选项的json文件来启动重新分配过程：

> bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --reassignment-json-file custom-reassignment.json --execute

Current partition replica assignment

{"version":1,

"partitions":[{"topic":"foo1","partition":0,"replicas":[1,2]},

{"topic":"foo2","partition":1,"replicas":[3,4]}]

}

Save this to use as the --reassignment-json-file option during rollback

Successfully started reassignment of partitions

{"version":1,

"partitions":[{"topic":"foo1","partition":0,"replicas":[5,6]},

{"topic":"foo2","partition":1,"replicas":[2,3]}]

}

--verify选项可与该工具一起使用，以检查分区重新分配的状态。请注意，相同的expand-cluster-reassignment.json（与--execute选项一起使用）应与--verify选项一起使用：

> bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --reassignment-json-file custom-reassignment.json --verify

Status of partition reassignment:

Reassignment of partition [foo1,0] completed successfully

Reassignment of partition [foo2,1] completed successfully

## 优雅的关机

Kafka群集将自动检测任何代理关闭或故障，并为该计算机上的分区选择新的领导者。无论服务器发生故障还是故意将其关闭以进行维护或配置更改，都会发生这种情况。对于后一种情况，Kafka支持更优雅的机制来停止服务器，而不仅仅是杀死服务器。当服务器正常停止时，它有两个优化：

1. 它会将所有日志同步到磁盘，以避免在重新启动时需要执行任何日志恢复（即验证日志尾部所有消息的校验和）。日志恢复需要时间，因此加速了故意重启。
2. 在关闭之前，它会将服务器所领先的任何分区迁移到其他副本。这将使领导层转移更快，并将每个分区不可用的时间缩短到几毫秒。

每当服务器停止而不是硬杀死时，将自动同步日志，但是leader迁移需要使用特殊的设置：

controlled.shutdown.enable=true

请注意，只有在代理上托管的所有分区都具有副本（即复制因子大于1 且这些副本中至少有一个处于活动状态）时，受控关闭才会成功。这通常是您想要的，因为关闭最后一个副本会使该主题分区不可用。

## 监控

虽然目前Apache Kafka已经全面进化成一个流处理平台，但大多数的用户依然使用的是其核心功能：消息队列。对于如何有效地监控和调优Kafka是一个大话题，很多用户都有这样的困扰，这章我们就来讨论一下。

当前没有一款Kafka监控工具是公认比较优秀的，每个都有自己的特点但也有些致命的缺陷。

主流的kafka监控工具有：Kafka Manager，Kafka Web Console，Burrow，Kafka Monitor，Kafka Offset Monitor，Kafka Eagle，Confluent Control Center。通过研究，发现主流的三种kafka监控程序分别为：

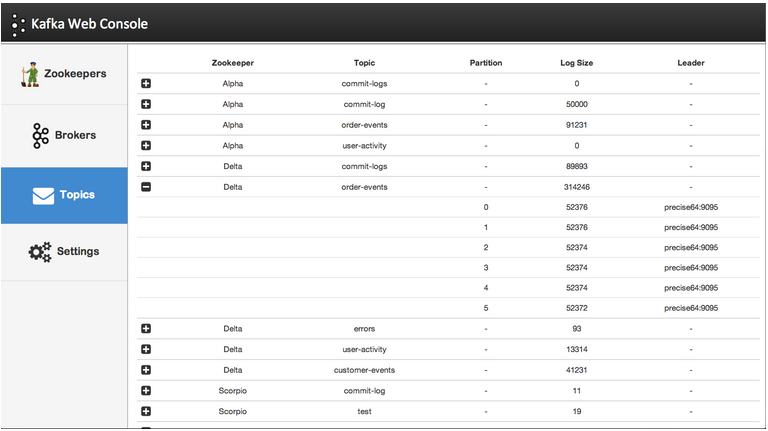
* Kafka Manager
* Kafka Offset Monitor
* Kafka Web Console

### Kafka Web Console

<https://github.com/claudemamo/kafka-web-console>

使用Kafka Web Console，可以监控：

* Brokers列表
* Kafka 集群中 Topic列表，及对应的Partition、LogSiz e等信息
* 点击Topic，可以浏览对应的Consumer Groups、Offset、Lag等信息
* 生产和消费流量图、消息预览…

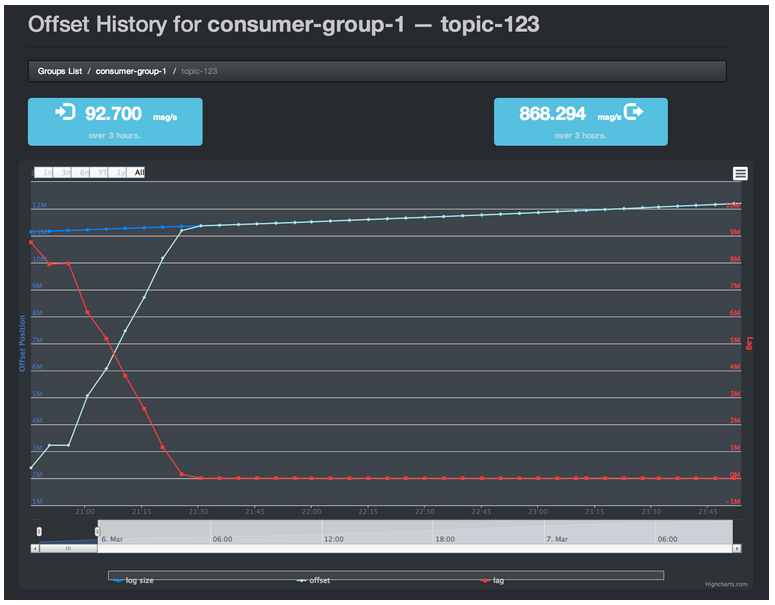


程序运行后，会定时去读取kafka集群分区的日志长度，读取完毕后，连接没有正常释放，一段时间后产生大量的socket连接，导致网络堵塞。

### Kafka Offset Monitor

Kafka Offset Monitor可以实时监控：

* Kafka集群状态
* Topic、Consumer Group列表
* 图形化展示topic和consumer之间的关系
* 图形化展示consumer的Offset、Lag等信息



总结:

通过使用，个人总结以上三种监控程序的优缺点：

Kafka Web Console：监控功能较为全面，可以预览消息，监控Offset、Lag等信息，但存在bug，不建议在生产环境中使用。

Kafka Manager：偏向Kafka集群管理，若操作不当，容易导致集群出现故障。对Kafka实时生产和消费消息是通过JMX实现的。没有记录Offset、Lag等信息。

KafkaOffsetMonitor：程序一个jar包的形式运行，部署较为方便。只有监控功能，使用起来也较为安全。

若只需要监控功能，推荐使用KafkaOffsetMonito，若偏重Kafka集群管理，推荐使用Kafka Manager。

因为都是开源程序，稳定性欠缺。故需先了解清楚目前已存在哪些Bug，多测试一下，避免出现类似于Kafka Web Console的问题。

---

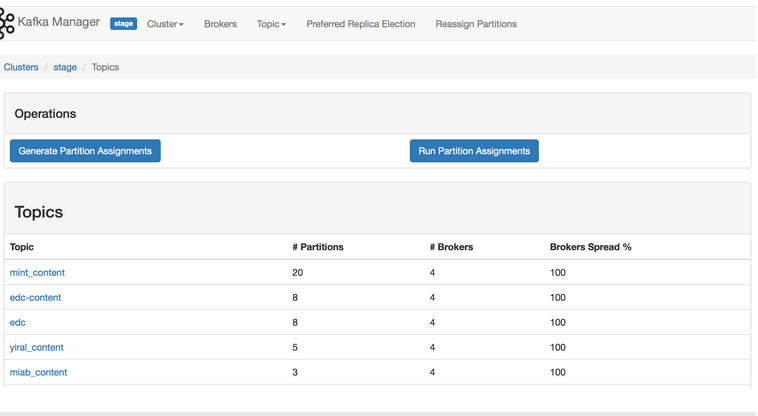
综合来讲，若只需要监控功能，推荐使用KafkaOffsetMonito，若偏重Kafka集群管理，推荐使用Kafka Manager。

### Kafka Manager

<https://github.com/yahoo/kafka-manager>

雅虎开源的Kafka集群管理工具:

* 管理几个不同的集群
* 监控集群的状态(topics, brokers, 副本分布, 分区分布)
* 产生分区分配(Generate partition assignments)基于集群的当前状态
* 重新分配分区



* **下载安装**

1. 安装sbt

yum install sbt

如果不行，则

curl https://bintray.com/sbt/rpm/rpm > bintray-sbt-rpm.repo

sudo mv bintray-sbt-rpm.repo /etc/yum.repos.d/

sudo yum install sbt

sbt 二进制文件发布到 Bintray，而Bintray 方便地提供了RPM资源库。你只需要将存储库添加到你的软件包管理器将检查的地方。

1. 下载编译

git clone https://github.com/yahoo/kafka-manager

cd kafka-manager

# 因为要编译。所以下面这步操作要等很久

sbt clean distcd target/

# 在target目录下我们可以看到 kafka-manager

kafka-manager-1.3.0.8.zip

提示：

使用sbt编译打包的时候时间可能会比较长，如果你hang在

Loading project definition from kafka-manager/project

可以修改project/plugins.sbt中的LogLevel参数

将logLevel := Level.Warn修改为logLevel := Level.Debug

1. 解压

unzip kafka-manager-1.3.0.8.zip -d /usr/local

cd /usr/local/kafka-manager-1.3.0.8

1. 修改配置conf/application.properties

# 如果zk是集群，这里填写多个zk地址

kafka-manager.zkhosts="localhost:2181"

1. 启动

bin/kafka-manager

kafka-manager 默认的端口是9000，可通过 -Dhttp.port，指定端口; -Dconfig.file=conf/application.conf指定配置文件:

nohup bin/kafka-manager -Dconfig.file=conf/application.conf -Dhttp.port=8080 &

kafka 默认是不开启JMX监控的，但是kafka-manager支持JMX监控，如果不添加，无法监控，所以我们需要配置kafka的JMX端口，并重启kafka

修改bin/kafka-server-start.sh，添加JMX\_PORT参数，添加后样子如下:

if [ "x$KAFKA\_HEAP\_OPTS" = "x" ]; then

export KAFKA\_HEAP\_OPTS="-Xmx1G -Xms1G"

export JMX\_PORT="9999"

Fi

这样便安装成功了。

## 调优

Kafka监控的一个主要的目的就是调优Kafka集群。这里罗列了一些常见的操作系统级的调优。

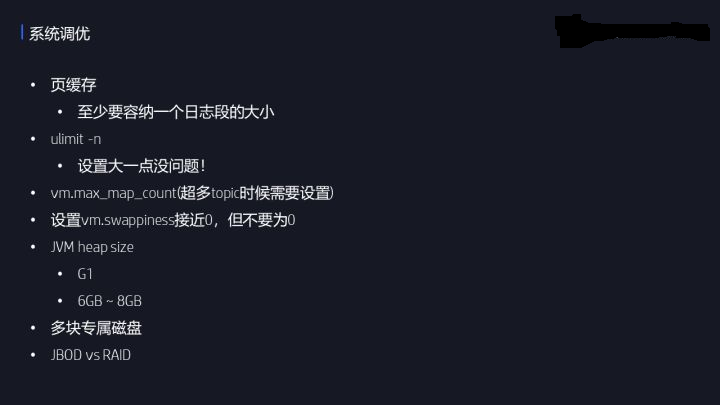
**首先是保证页缓存的大小——至少要设置页缓存为一个日志段的大小。**我们知道Kafka大量使用页缓存，只要保证页缓存足够大，那么消费者读取消息时就有大概率保证它能够直接命中页缓存中的数据而无需从底层磁盘中读取。故只要保证页缓存要满足一个日志段的大小。

**第二是调优文件打开数。很多人对这个资源有点畏手畏脚。**实际上这是一个很廉价的资源，设置一个比较大的初始值通常都是没有什么问题的。

**第三是调优vm.max\_map\_count参数。**主要适用于Kafka broker上的主题数超多的情况。Kafka日志段的索引文件是用映射文件的机制来做的，故如果有超多日志段的话，这种索引文件数必然是很多的，极易打爆这个资源限制，所以对于这种情况一般要适当调大这个参数。

**第四是swap的设置。**很多文章说把这个值设为0，就是完全禁止swap，我个人不建议这样，因为当你设置成为0的时候，一旦你的内存耗尽了，Linux会自动开启OOM killer然后随机找一个进程杀掉。这并不是我们希望的处理结果。相反，我建议设置该值为一个比较接近零的较小值，这样当我的内存快要耗尽的时候会尝试开启一小部分swap，虽然会导致broker变得非常慢，但至少给了用户发现问题并处理之的机会。

**第五JVM堆大小。**首先鉴于目前Kafka新版本已经不支持Java7了，而Java 8本身不更新了，甚至Java9其实都不做了，直接做Java10了，所以我建议Kafka至少搭配Java8来搭建。至于堆的大小，个人认为6-10G足矣。如果出现了堆溢出，就提jira给社区，让他们看到底是怎样的问题。因为这种情况下即使用户调大heap size，也只是延缓OOM而已，不太可能从根本上解决问题。



最后，建议使用专属的多块磁盘来搭建Kafka集群。自1.1版本起Kafka正式支持JBOD，因此没必要在底层再使用一套RAID了。

## Kafka配置信息

### 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. |
| 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. |

### 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. |
| 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. |

### 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. |
| 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 |

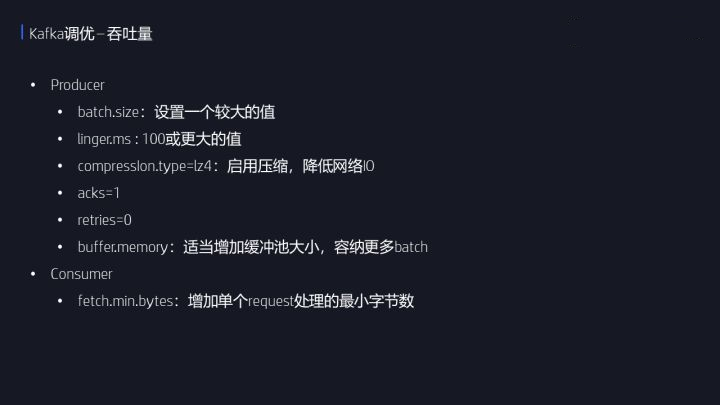
## Kafka调优的四个层面

Kafka调优通常可以从4个维度展开，分别是吞吐量、延迟、持久性和可用性。在具体展开这些方面之前，我想先建议用户保证客户端与服务器端版本一致。如果版本不一致，就会出现向下转化的问题。举个例子，服务器端保存高版本的消息，当低版本消费者请求数据时，服务器端就要做转化，先把高版本消息转成低版本再发送给消费者。这件事情本身就非常非常低效。很多文章都讨论过Kafka速度快的原因，其中就谈到了零拷贝技术——即数据不需要在页缓存和堆缓存中来回拷贝。

简单来说producer把生产的消息放到页缓存上，如果两边版本一致，可以直接把此消息推给Consumer，或者Consumer直接拉取，这个过程是不需要把消息再放到堆缓存。但是你要做向下转化或者版本不一致的话，就要额外把数据再堆上，然后再放回到Consumer上，速度特别慢。

### Kafka调优-吞吐量

调优吞吐量就是我们想用更短的时间做更多的事情。这里列出了客户端需要调整的参数。前面说过了producer是把消息放在缓存区，后端Sender线程从缓存区拿出来发到broker。这里面涉及到一个打包的过程，它是批处理的操作，不是一条一条发送的。因此这个包的大小就和TPS息息相关。通常情况下调大这个值都会让TPS提升，但是也不会无限制的增加。不过调高此值的劣处在于消息延迟的增加。除了调整batch.size，设置压缩也可以提升TPS，它能够减少网络传输IO。当前Lz4的压缩效果是最好的，如果客户端机器CPU资源很充足那么建议开启压缩。



对于消费者端而言，调优TPS并没有太好的办法，能够想到的就是调整fetch.min.bytes。适当地增加该参数的值能够提升consumer端的TPS。对于Broker端而言，通常的瓶颈在于副本拉取消息时间过长，因此可以适当地增加num.replica.fetcher值，利用多个线程同时拉取数据，可以加快这一进程。

### Kafka调优-延时

所谓的延时就是指消息被处理的时间。某些情况下我们自然是希望越快越好。针对这方面的调优，consumer端能做的不多，简单保持fetch.min.bytes默认值即可，这样可以保证consumer能够立即返回读取到的数据。讲到这里，可能有人会有这样的疑问：TPS和延时不是一回事吗？假设发一条消息延时是2ms，TPS自然就是500了，因为一秒只能发500消息，其实这两者关系并不是简单的。因为我发一条消息2毫秒，但是如果把消息缓存起来统一发，TPS会提升很多。假设发一条消息依然是2ms，但是我先等8毫秒，在这8毫秒之内可能能收集到一万条消息，然后我再发。相当于你在10毫秒内发了一万条消息，大家可以算一下TPS是多少。事实上，Kafka producer在设计上就是这样的实现原理。

### Kafka调优-消息持久性

消息持久化本质上就是消息不丢失。Kafka对消息不丢失的承诺是有条件的。以前碰到很多人说我给Kafka发消息，发送失败，消息丢失了，怎么办？严格来说Kafka不认为这种情况属于消息丢失，因为此时消息没有放到Kafka里面。Kafka只对已经提交的消息做有条件的不丢失保障。

如果要调优持久性，对于producer而言，首先要设置重试以防止因为网络出现瞬时抖动造成消息发送失败。一旦开启了重试，还需要防止乱序的问题。比如说我发送消息1与2，消息2发送成功，消息1发送失败重试，这样消息1就在消息2之后进入Kafka，也就是造成乱序了。如果用户不允许出现这样的情况，那么还需要显式地设置max.in.flight.requests.per.connection为1。



其他参数都是很常规的参数，比如unclean.leader.election.enable参数，最好还是将其设置成false，即不允许“脏”副本被选举为leader。

### Kafka调优-可用性

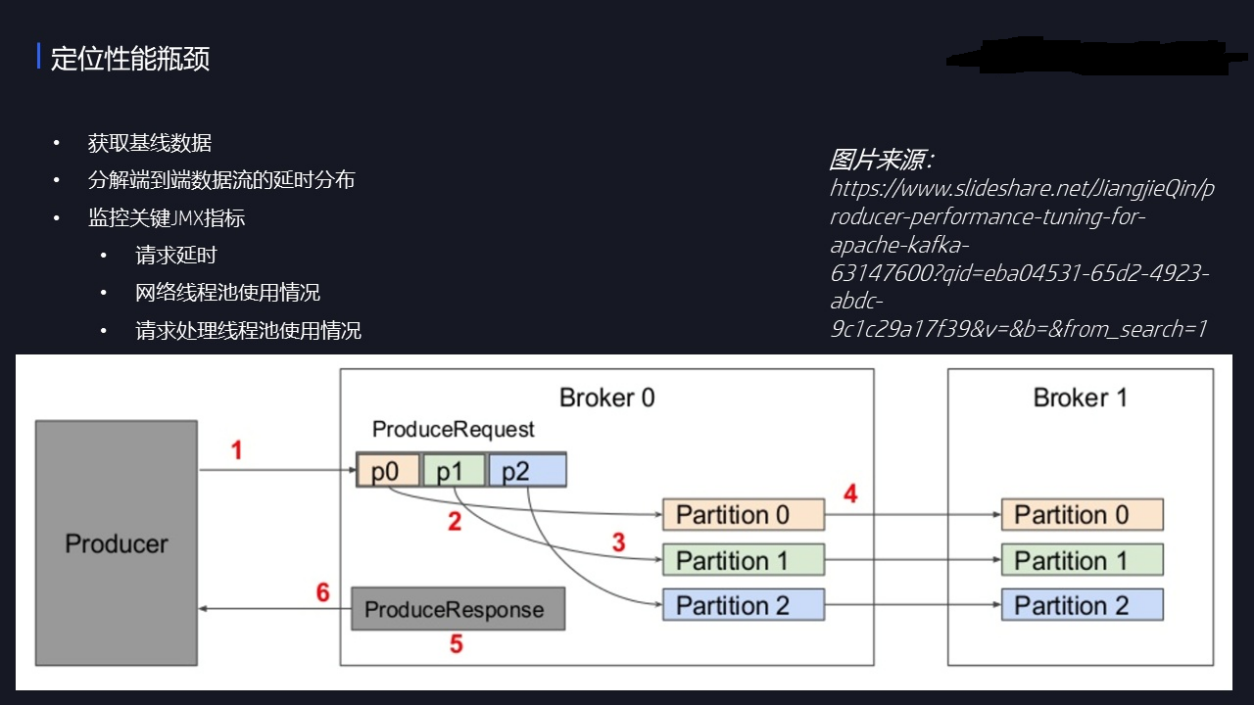
最后是可用性，与刚才的持久性是相反的，我允许消息丢失，只要保证系统高可用性即可。因此我需要把consumer心跳超时设置为一个比较小的值，如果给定时间内消费者没有处理完消息，该实例可能就被踢出消费者组。我想要其他消费者更快地知道这个决定，因此调小这个参数的值。

## 定位性能瓶颈

下面就是性能瓶颈，严格来说这不是调优，这是解决性能问题。对于生产者来说，如果要定位发送消息的瓶颈很慢，我们需要拆解发送过程中的各个步骤。就像这张图表示的那样，消息的发送共有6步。

第一步就是生产者把消息放到Broker，

1. 三步就是Broker把消息拿到之后，写到本地磁盘上，
2. 第四步是follower broker从Leader拉取消息，
3. 第五步是创建response，
4. 第六步是发送回去，告诉我已经处理完了。



这六步当中你需要确定瓶颈在哪？怎么确定？

——

通过不同的JMX指标。

比如说步骤1是慢的，可能你经常碰到超时，你如果在日志里面经常碰到request timeout，就表示1是很慢的，此时要适当增加超时的时间。

如果2、3慢的情况下，则可能体现在磁盘IO非常高，导致往磁盘上写数据非常慢。

倘若是步骤4慢的话，查看名为remote-time的JMX指标，此时可以增加fetcher线程的数量。

如果5慢的话，表现为response在队列导致待的时间过长，这时可以增加网络线程池的大小。

6与1是一样的，如果你发现1、6经常出问题的话，查一下你的网络。

所以，就这样来分解整个的耗时。这是到底哪一步的瓶颈在哪，需要看看什么样的指标，做怎样的调优。

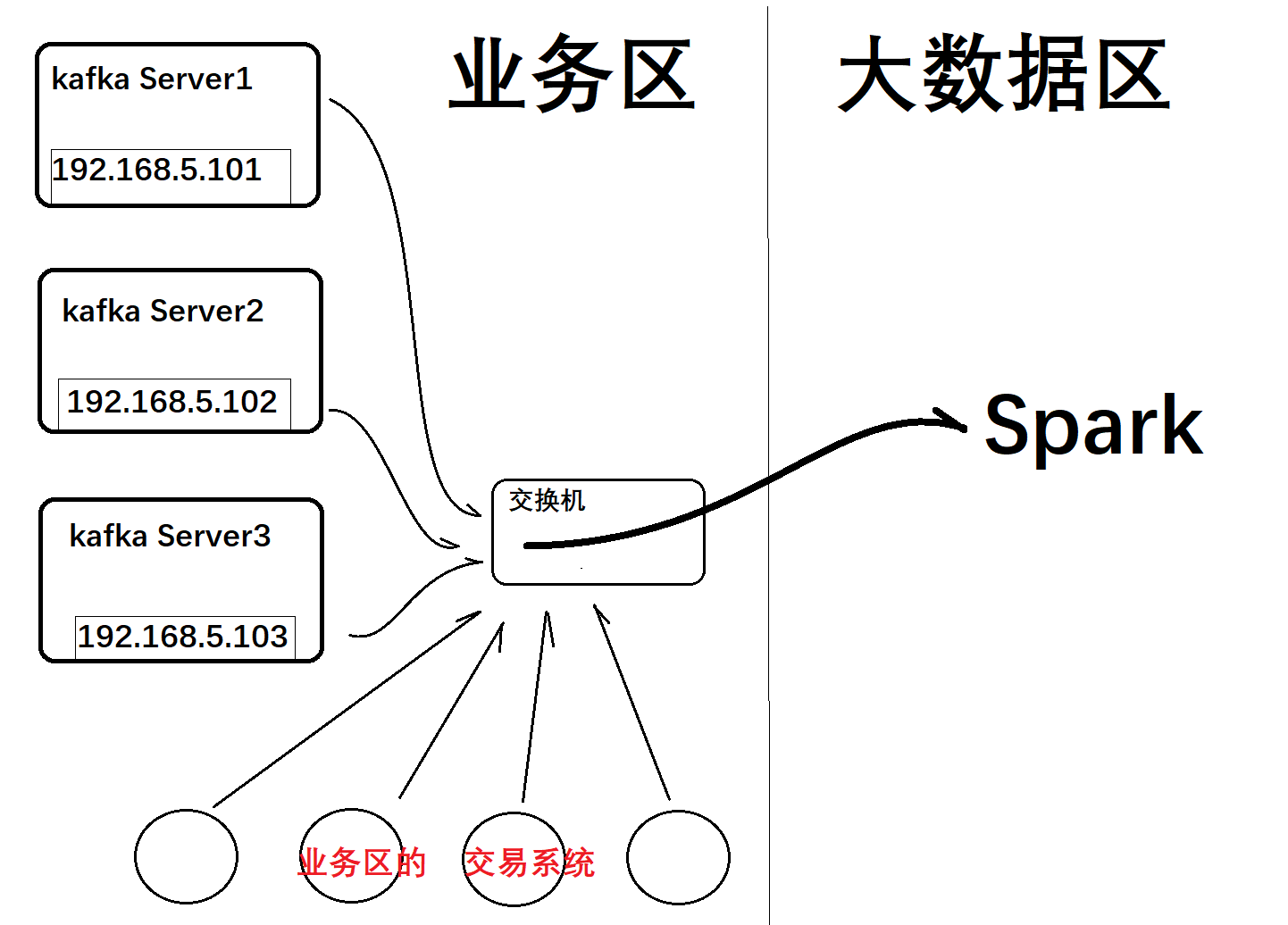
## Java Consumer调优

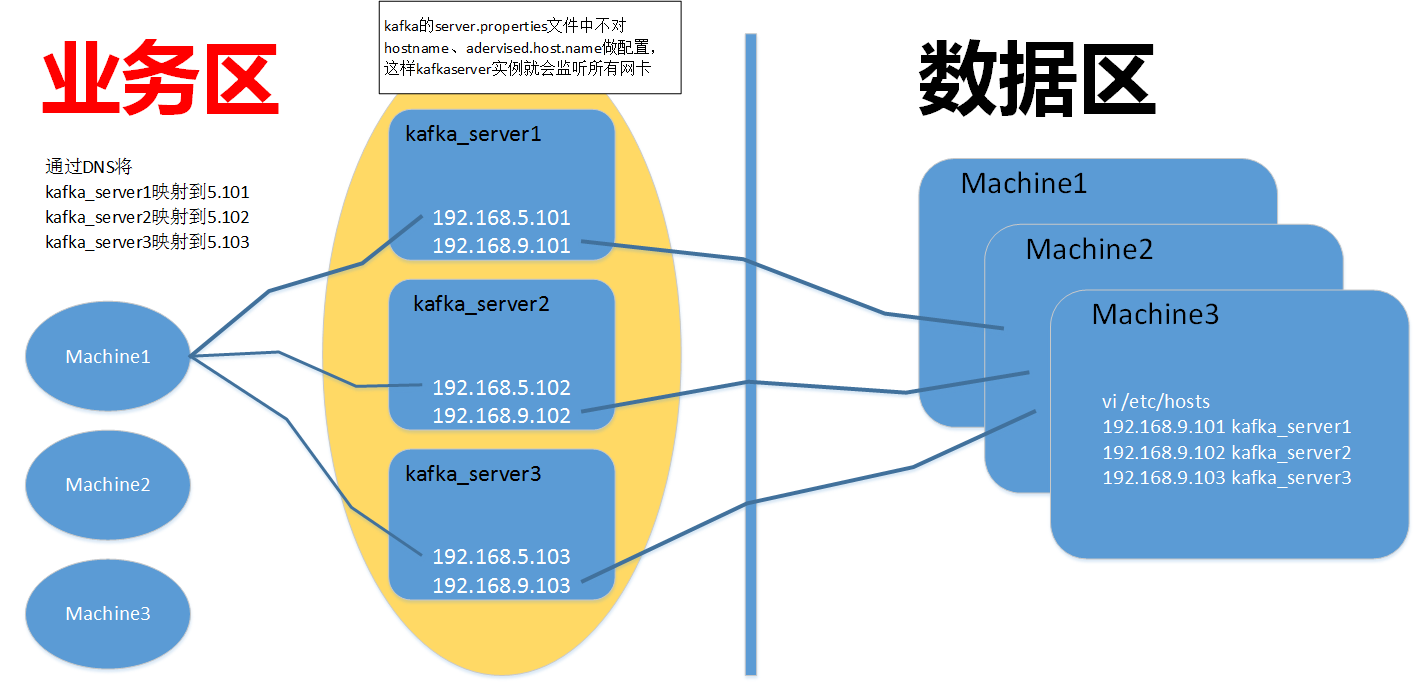
最后说一下Consumer的调优。目前消费者有两种使用方式，一种是同一个线程里面就直接处理，另一种是我采用单独的线程，consumer线程只是做获取消息，消息真正的处理逻辑放到单独的线程池中做。这两种方式有不同的使用场景：第一种方法实现较简单，因为你的消息处理逻辑直接写在一个线程里面就可以了，但是它的缺陷在于TPS可能不会很高，特别是当你的客户端的机器非常强的时候，你用单线程处理的时候是很慢的，因为你没有充分利用线程上的CPU资源。第二种方法的优势是能够充分利用底层服务器的硬件资源，TPS可以做的很高，但是处理提交位移将会很难。

最后说一下参数，也是网上问的最多的，这几个参数到底是做什么的。第一个参数，就是控制consumer单次处理消息的最大时间。比如说设定的是600s，那么consumer给你10分钟来处理。如果10分钟内consumer无法处理完成，那么coordinator就会认为此consumer已死，从而开启rebalance。

Coordinator是用来管理消费者组的协调者，协调者如何在有效的时间内，把消费者实例挂掉的消息传递给其他消费者，就靠心跳请求，因此可以设置heartbeat.interval.ms为一个较小的值，比如5s。

## 流量激增和多网卡方案





## Kafka命令大全

整理kafka相关的常用命令

### 管理

## 创建主题（4个分区，2个副本）

bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 2 --partitions 4 --topic test

注： partitions指定topic分区数，replication-factor指定topic每个分区的副本数

* partitions分区数：
* partitions ：分区数，控制topic将分片成多少个log。可以显示指定，如果不指定则会使用broker(server.properties)中的num.partitions配置的数量
* 虽然增加分区数可以提供kafka集群的吞吐量、但是过多的分区数或者或是单台服务器上的分区数过多，会增加不可用及延迟的风险。因为多的分区数，意味着需要打开更多的文件句柄、增加点到点的延时、增加客户端的内存消耗。
* 分区数也限制了consumer的并行度，即限制了并行consumer消息的线程数不能大于分区数
* 分区数也限制了producer发送消息是指定的分区。如创建topic时分区设置为1，producer发送消息时通过自定义的分区方法指定分区为2或以上的数都会出错的；这种情况可以通过alter –partitions 来增加分区数。
* replication-factor副本
* replication factor 控制消息保存在几个broker(服务器)上，一般情况下等于broker的个数。
* 如果没有在创建时显示指定或通过API向一个不存在的topic生产消息时会使用broker(server.properties)中的default.replication.factor配置的数量

## 查询集群描述

bin/kafka-topics.sh --describe --zookeeper

## 增加分区：为topic test 增加10个分区

bin/kafka-topics.sh --zookeeper 192.168.100.249:2181 --alter --topic test --partitions 10

## 删除topic，慎用，只会删除zookeeper中的元数据，消息文件须手动删除

bin/kafka-run-class.sh kafka.admin.DeleteTopicCommand --zookeeper 192.168.100.249:2181 --topic test

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

注意：如果delete.topic.enable未设置为true，则不会对数据产生影响。

### 查询

## 查询集群描述

bin/kafka-topics.sh --describe --zookeeper

## 消费者列表查询

bin/kafka-topics.sh --zookeeper 127.0.0.1:2181 --list

## 新消费者列表查询（支持0.9版本+）

bin/kafka-consumer-groups.sh --new-consumer --bootstrap-server localhost:9092 --list

## 显示某个消费组的消费详情（仅支持offset存储在zookeeper上的）

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

## 显示某个消费组的消费详情（支持0.9版本+）

bin/kafka-consumer-groups.sh --new-consumer --bootstrap-server localhost:9092 --describe --group test-consumer-group

## 查看topic某分区偏移量最大（小）值

bin/kafka-run-class.sh kafka.tools.GetOffsetShell --topic test --time -1 --broker-list 192.168.100.249:9092 --partitions 0

注： time为-1时表示最大值，time为-2时表示最小值

### 发送和消费

## 生产者

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

## 消费者

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

## 新生产者（支持0.9版本+）

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

## 新消费者（支持0.9版本+）

bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic test --new-consumer --from-beginning --consumer.config config/consumer.properties

## 高级点的用法

bin/kafka-simple-consumer-shell.sh --brist localhost:9092 --topic test --partition 0 --offset 1234 --max-messages 10

### 平衡leader

bin/kafka-preferred-replica-election.sh --zookeeper 192.168.100.249:2181

### Kafka自带压测命令

bin/kafka-producer-perf-test.sh --topic test --num-records 100 --record-size 1 --throughput 100 --producer-props bootstrap.servers=192.168.100.249:9092

### 增加副本

1.创建规则json

cat > increase-replication-factor.json <<EOF

{"version":1, "partitions":[

{"topic":"\_\_consumer\_offsets","partition":0,"replicas":[0,1]},

{"topic":"\_\_consumer\_offsets","partition":1,"replicas":[0,1]},

{"topic":"\_\_consumer\_offsets","partition":2,"replicas":[0,1]},

{"topic":"\_\_consumer\_offsets","partition":3,"replicas":[0,1]},

{"topic":"\_\_consumer\_offsets","partition":4,"replicas":[0,1]},

{"topic":"\_\_consumer\_offsets","partition":5,"replicas":[0,1]},

{"topic":"\_\_consumer\_offsets","partition":6,"replicas":[0,1]},

{"topic":"\_\_consumer\_offsets","partition":7,"replicas":[0,1]},

{"topic":"\_\_consumer\_offsets","partition":8,"replicas":[0,1]},

{"topic":"\_\_consumer\_offsets","partition":9,"replicas":[0,1]}]

}

EOF

1. 执行

bin/kafka-reassign-partitions.sh --zookeeper 192.168.100.249:2181 --reassignment-json-file increase-replication-factor.json --execute

3.验证

bin/kafka-reassign-partitions.sh --zookeeper 192.168.100.249:2181 --reassignment-json-file increase-replication-factor.json --verify