Kafka介绍&

安装部署

目录

[1 Kafka介绍 4](#_Toc476132447)

[1.1 Kafka定义 4](#_Toc476132448)

[1.2 Kafka技术概览 4](#_Toc476132449)

[1.2.1 Kafka的特性 4](#_Toc476132450)

[1.2.2 Kafka一些重要设计思想 4](#_Toc476132451)

[1.2.3 Kafka Topic 5](#_Toc476132452)

[1.2.4 Kafka的Log存储解析 6](#_Toc476132453)

[1.3 Kafka 核心组件 9](#_Toc476132454)

[1.3.1 Partitions 9](#_Toc476132455)

[1.3.2 Replications 9](#_Toc476132456)

[1.3.3 Producers 10](#_Toc476132457)

[1.3.4 Consumers 11](#_Toc476132458)

[1.4 Kafka主要配置 12](#_Toc476132459)

[1.4.1 Broker配置 12](#_Toc476132460)

[1.4.2 Producer（生产者）配置 15](#_Toc476132461)

[1.4.3 Consumer（消费者）配置 16](#_Toc476132462)

[2 Zookeeper的介绍和安装 18](#_Toc476132463)

[2.1 zookeeper介绍 18](#_Toc476132464)

[2.2 Zookeeper在kafka中的应用 18](#_Toc476132465)

[2.3 安装部署zookeeper 19](#_Toc476132466)

[2.3.1 安装zookeeper 19](#_Toc476132467)

[2.3.2 启动zookeeper 20](#_Toc476132468)

[2.4 Zookeeper集群配置【伪集群】 21](#_Toc476132469)

[2.4.1 部署规模为3的单机伪机群 21](#_Toc476132470)

[2.4.2 配置 21](#_Toc476132471)

[2.4.3 启动 22](#_Toc476132472)

[2.4.4 集群状态查询 23](#_Toc476132473)

[3 安装部署kafka 23](#_Toc476132474)

[3.1 环境配置 23](#_Toc476132475)

[3.2 安装Kafka 23](#_Toc476132476)

[3.2.1 下载Kafka并解压 23](#_Toc476132477)

[3.2.2 Kafka目录介绍 24](#_Toc476132478)

[3.2.3 修改server.properties配置文件 24](#_Toc476132479)

[3.3 启动kafka 24](#_Toc476132480)

[3.3.1 启动命令 24](#_Toc476132481)

[3.3.2 检测2181与9092端口 24](#_Toc476132482)

[3.3.3 连通性测试 25](#_Toc476132483)

[4 kafka集群配置【伪集群】 25](#_Toc476132484)

[4.1 部署3个broker的伪集群 25](#_Toc476132485)

[4.2 配置 26](#_Toc476132486)

[4.3 启动 26](#_Toc476132487)

[4.3.1 启动命令 26](#_Toc476132488)

[4.3.2 查看端口 27](#_Toc476132489)

[4.4 验证 27](#_Toc476132490)

[4.4.1 创建topic 27](#_Toc476132491)

[4.4.2 查看topic创建情况 27](#_Toc476132492)

[4.4.3 模拟客户端发送，接受消息 28](#_Toc476132493)

# Kafka介绍

## Kafka定义

Kafka是一个分布式消息系统，由linkedin使用scala编写，用作LinkedIn的活动流（Activity Stream）和运营数据处理管道（Pipeline）的基础。具有高水平扩展和高吞吐量。

生产者生产消息、kafka集群、消费者获取消息这样一种架构，如下图：



kafka对消息保存时根据Topic进行归类，发送消息者成为Producer,消息接受者成为Consumer,此外kafka集群有多个kafka实例组成，每个实例(server)成为broker。无论是kafka集群，还是producer和consumer都依赖于zookeeper来保证系统可用性集群保存一些meta信息。

## Kafka技术概览

### Kafka的特性

* 高吞吐量、低延迟：kafka每秒可以处理几十万条消息，它的延迟最低只有几毫秒
* 可扩展性：kafka集群支持热扩展
* 持久性、可靠性：消息被持久化到本地磁盘，并且支持数据备份防止数据丢失
* 容错性：允许集群中节点失败（若副本数量为n,则允许n-1个节点失败）
* 高并发：支持数千个客户端同时读写

### Kafka一些重要设计思想

先大体介绍一下Kafka的主要设计思想。

* **Consumergroup**：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。high-level consumer API 中，每个 consumer 都属于一个 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。详细信息参考 ***1.3.4 consumers*** 章节 。
* **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。具体参考 ***1.2.4.1存储方式*** 章节
* **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。具体参考 ***1.3.3.2 异步发送*** 章节。
* **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。具体参考 ***1.3.3.2 异步发送*** 章节。
* **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）。具体参考 ***1.3.3 Producers*** 章节。
* **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。具体参考 ***1.3.1 Partitions*** 章节。

### Kafka Topic

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



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。

Kafka集群会保存所有的消息，不管消息有没有被消费；我们可以设定消息的过期时间，只有过期的数据才会被自动清除以释放磁盘空间。比如我们设置消息过期时间为2天，那么这2天内的所有消息都会被保存到集群中，数据只有超过了两天才会被清除。

Kafka需要维持的元数据只有一个消费消息在Partition中的offset值，Consumer每消费一个消息，offset就会加1。其实消息的状态完全是由Consumer控制的，Consumer可以跟踪和重设这个offset值，这样的话Consumer就可以读取任意位置的消息。

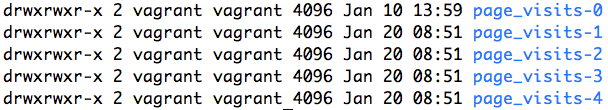
把消息日志以Partition的形式存放有多重考虑，第一，方便在集群中扩展，每个Partition可以通过调整以适应它所在的机器，而一个topic又可以有多个Partition组成，因此整个集群就可以适应任意大小的数据了；第二就是可以提高并发，因为可以以Partition为单位读写了。

### Kafka的Log存储解析

#### 存储方式

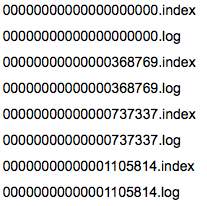
partition是以文件的形式存储在文件系统中，比如，创建了一个名为page\_visits的topic，其有5个partition，那么在Kafka的数据目录中(由配置文件中的log.dirs指定的)中就有这样5个目录：page\_visits-0、page\_visits-1、page\_visits-2、page\_visits-3、page\_visits-4，其命名规则为<topic\_name>-<partition\_id>，里面存储的分别就是这5个partition的数据。

其目录结构如下：



#### 数据文件和索引

每一个partition当中，又分成了多个segment，每个segment包括了2个文件：index文件和log文件，分别是消息的索引信息和消息内容。下图是某个partition目录下的文件：



可以看到，这个partition有4个Segment。文件名就是消息在本分区当中的序号，以上面的文件为例，第一个segment当中存储了0-368769的消息，而第二个segment中存储了368770-737337的消息，第三个segment中存储了737338-1105814的消息，相应的第四个segment从1105815开始。

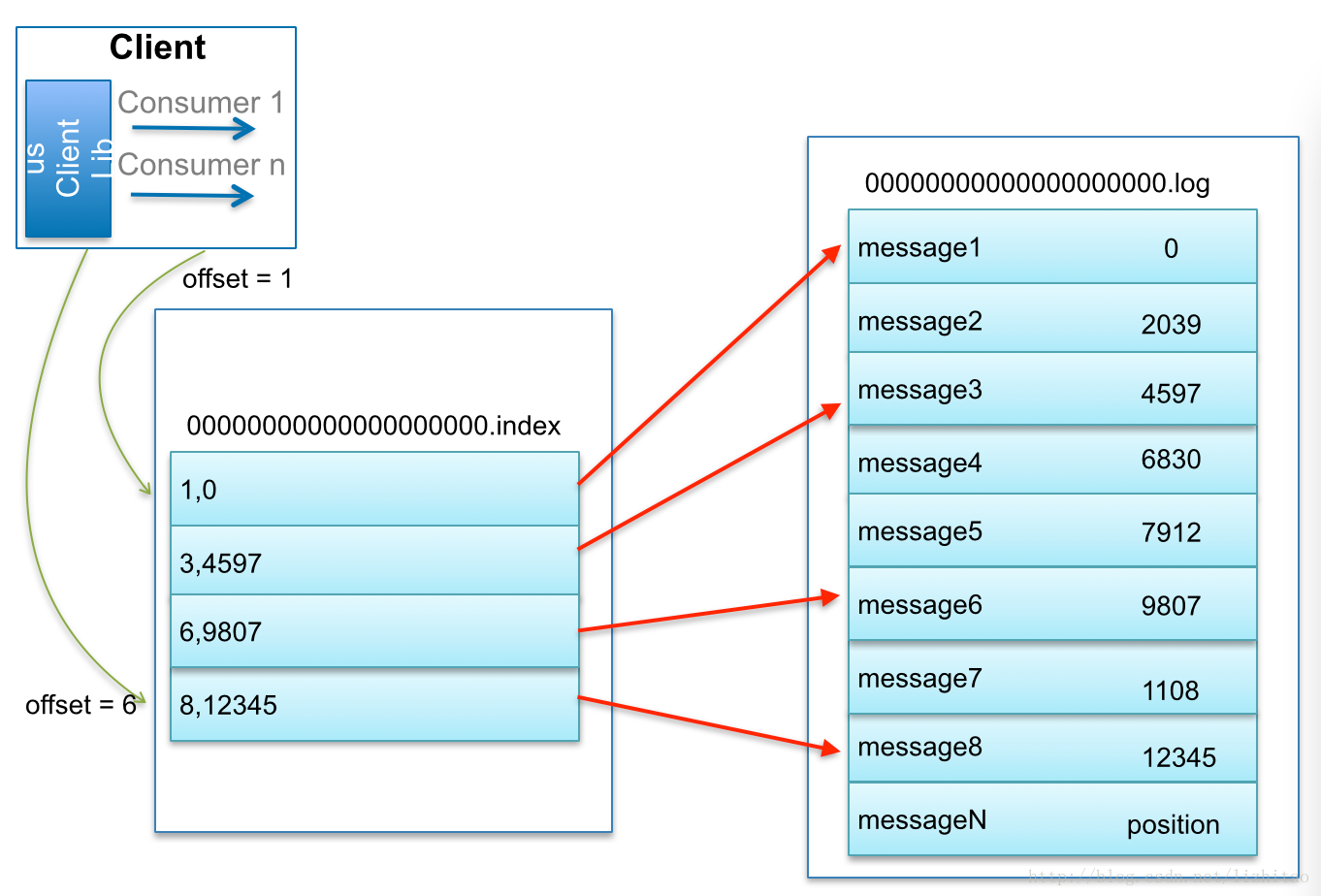
index文件当中，包含了2部分内容，分别是消息在本segment当中的相对offset和position。

* **相对offset**：因为数据文件分段以后，每个数据文件的起始offset不为0，相对offset表示这条Message相对于其所属数据文件中最小的offset的大小。举例，分段后的一个数据文件的offset是从20开始，那么offset为25的Message在index文件中的相对offset就是25-20 = 5。存储相对offset可以减小索引文件占用的空间。
* **Position**：表示该条Message在数据文件中的绝对位置。只要打开文件并移动文件指针到这个position就可以读取对应的Message了。

这里需要注意的是，在index文件当中，并不是存储了每一条消息的索引信息，而是采用了稀疏索引的策略，也就是隔几个存一个索引。

#### 查找消息的方式

查找一条消息的时候，首先是根据segment的名字和输入的offset，查找所在的segment，之后在根据index文件找到消息的具体位置。下图展示如何查找Message的：



查找绝对offset为7的Message：

1. 首先用“二分查找法”确定它是在哪个LogSegment中，找到第一个Segment中。
2. 打开这个Segment的index文件(..0000.index)，也用二分查找法找到offset小于或者等于指定offset的索引条目中最大的那个offset。自然offset为6的那个索引是我们要找的，通过索引文件我们知道offset为6的Message在数据文件中的位置为9807。
3. 打开数据文件，从位置为9807的那个地方开始顺序扫描直到找到offset为7的那条Message。

#### 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：

* 基于时间：log.retention.hours=168
* 基于大小：log.retention.bytes=1073741824

这2个配置可在 ***1.4.1Broker配置*** 章节查看详细信息。

## Kafka 核心组件

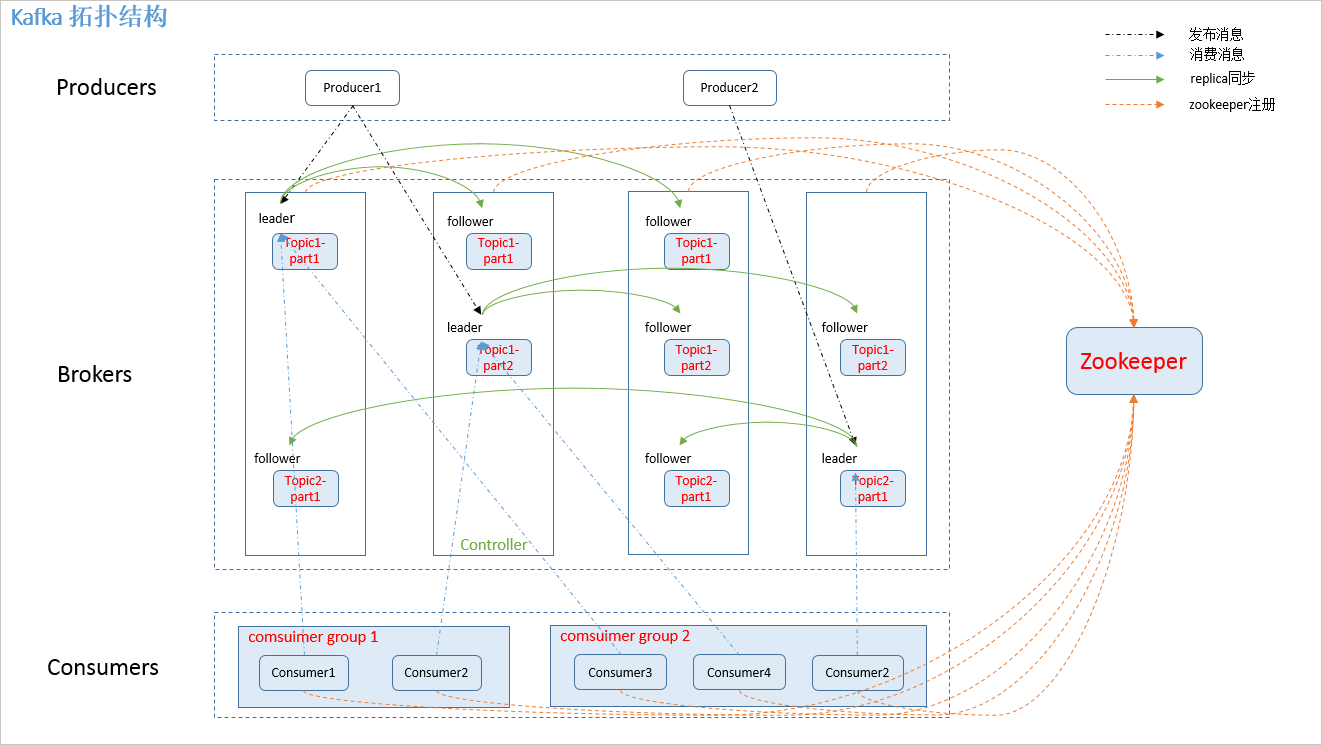


图 1 kafka拓扑图

### Partitions

Kafka中的topic是以partition的形式存放的，每一个topic都可以设置它的partition数量，Partition的数量决定了组成topic的log的数量。Producer在生产数据时，会按照一定规则（这个规则是可以自定义的）把消息发布到topic的各个partition中。**拓扑图**中，topic1被分为2个partition：topic1-part1和topic1-part2。

如何设置partition值需要考虑的因素。一个partition只能被一个消费者消费（一个消费者可以同时消费多个partition），因此，如果设置的partition的数量小于consumer的数量，就会有消费者消费不到数据。所以，推荐partition的数量一定要大于同时运行的consumer的数量。另外一方面，建议partition的数量大于集群broker的数量，这样leader partition就可以均匀的分布在各个broker中，最终使得集群负载均衡。在Cloudera,每个topic都有上百个partition。需要注意的是，kafka需要为每个partition分配一些内存来缓存消息数据，如果partition数量越大，就要为kafka分配更大的heap space。

### Replications

Replication 是partition 的副本，用来保障 partition 的高可用。副本以partition为单位进行存储，只有一个partition的副本会被选举成leader作为读写用。

Kafka允许用户为每个topic设置副本数量，副本数量决定了有几个broker来存放写入的数据。如果你的副本数量设置为3，那么一份数据就会被存放在3台不同的机器上，那么就允许有2个机器失败。一般推荐副本数量至少为2，这样就可以保证增减、重启机器时不会影响到数据消费。如果对数据持久化有更高的要求，可以把副本数量设置为3或者更多。**拓扑图**中，topic1-part1有3个副本，其中一个是 **leader**，另外2个是**follower**。Consumer和producers只跟leader进行信息交互。Follower则从 leader 中复制数据，进行数据备份。

### Producers

Producers直接发送消息到broker上的leader partition，不需要经过任何中介一系列的路由转发。为了实现这个特性，kafka集群中的每个broker都可以响应producer的请求，并返回topic的一些元信息，这些元信息包括哪些机器是存活的，topic的leader partition都在哪，现阶段哪些leader partition是可以直接被访问的。

Producer客户端自己控制着消息被推送到哪些partition。实现的方式可以是随机分配、实现一类随机负载均衡算法，或者指定一些分区算法。Kafka提供了接口供用户实现自定义的分区，用户可以为每个消息指定一个partitionKey，通过这个key来实现一些hash分区算法。比如，把userid作为partitionkey的话，相同userid的消息将会被推送到同一个分区。

Kafka 消息由一个定长的header和变长的字节数组组成。因为kafka消息支持字节数组，也就使得kafka可以支持任何用户自定义的序列号格式或者其它已有的格式如Apache Avro、protobuf等。Kafka没有限定单个消息的大小，但我们推荐消息大小不要超过1MB，通常一般消息大小都在1~10kB之前。

kafka producer 端发送数据分为同步（实时）和异步（达到设定发送条件）。

#### 同步发送

kafka在同步（sync）模式下发送消息时，把message发送给broker，可以是一条消息也可以是多条消息组成的List。异步发送相关参数如下：

producer.type=sync #producer设置为同步模式

#### 异步发送

异步发送模式允许通过Batch（批量发送）的方式推送数据，可以极大的提高处理效率（推荐使用异步模式），kafka Producer 可以将消息在内存中累计到一定数量后作为一个batch发送请求。

Batch的数量大小可以通过Producer的参数控制，参数值可以设置为累计的消息的数量（如500条）、累计的时间间隔（如100ms）或者累计的数据大小(64KB)。通过增加batch的大小，可以减少网络请求和磁盘IO的次数，当然具体参数设置需要在效率和时效性方面做一个权衡。异步发送相关参数如下：

producer.type=async #producer设置为异步模式

当满足以下其中一个条件的时候就触发发送

batch.num.messages 异步发送时每次批量发送的条目

queue.buffering.max.ms 异步发送时的发送时间间隔，单位是毫秒

queue.buffering.max.messages采用异步模式时最大缓存的消息数量

### Consumers

Kafka提供了两套consumer api，分为high-level api和sample-api。Sample-api 是一个底层的API，它维持了一个和单一broker的连接，并且这个API是完全无状态的，每次请求都需要指定offset值，因此，这套API也是最灵活的。

在kafka中，当前读到消息的offset值是由consumer来维护的，因此，consumer可以自己决定如何读取kafka中的数据。比如，consumer可以通过重设offset值来重新消费已消费过的数据。不管有没有被消费，kafka会保存数据一段时间，这个时间周期是可配置的，只有到了过期时间，kafka才会删除这些数据。

High-level API封装了对集群中一系列broker的访问，可以透明的消费一个topic。它自己维持了已消费消息的状态，即每次消费的都是下一个消息。

High-level API还支持以组的形式消费topic，如果consumers有同一个组名，那么kafka就相当于一个队列消息服务，而各个consumer均衡的消费相应partition中的数据。若consumers有不同的组名，那么此时kafka就相当与一个广播服务，会把topic中的所有消息广播到每个consumer。



一个**Consumergroup**(消费者组)可以有多个消费者，Kafka中的一个Partition只会被消费者组中的一个消费者消费，但可以被多个消费组同时消费。

图中Server1的Partition0(P0)，会被消费组A和消费组B同时消费，但是消费组A中只能有一个消费者读取比如C1(消费组B也只有C3读取)。

通过这种方式，可以将一个Broker上的多个Partition负载到不同的消费者(同一个消费组的多个消费者)。比如Broker1上有两个Partition(P0,P3)， 对于消费组B，P0分给C3,P3分给C4。但是对于消费组A，都分给了C1。

## Kafka主要配置

### Broker配置

|  |  |  |
| --- | --- | --- |
| 属性 | 默认值 | 描述 |
| broker.id | 0 | 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（生产者）配置

|  |  |  |
| --- | --- | --- |
| 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 | 这个参数决定了producer要求leader partition 收到确认的副本个数，如果acks设置数量为0，表示producer不会等待broker的响应，所以，producer无法知道消息是否发送成功，这样有可能会导致数据丢失，但同时，acks值为0会得到最大的系统吞吐量。  若acks设置为1，表示producer会在leader partition收到消息时得到broker的一个确认，这样会有更好的可靠性，因为客户端会等待直到broker确认收到消息。若设置为-1，producer会在所有备份的partition收到消息时得到broker的确认，这个设置可以得到最高的可靠性保证。 |
| 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”。 |
| 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 | 缓存区大小 |
| 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（消费者）配置

|  |  |  |
| --- | --- | --- |
| 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。 |
| rebalance.max.retries | 4 | When a new consumer joins a consumer group the set of consumers attempt to “rebalance” the load to assign partitions to each consumer. If the set of consumers changes while this assignment is taking place the rebalance will fail and retry. This setting controls the maximum number of attempts before giving up. |
| 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 |

以上是关于kafka一些基础说明，在其中我们知道如果要kafka正常运行，必须配置zookeeper，否则无论是kafka集群还是客户端的生存者和消费者都无法正常的工作的，以下是对zookeeper进行一些简单的介绍：

# Zookeeper的介绍和安装

## zookeeper介绍

Zookeeper是一个分布式开源框架，提供了协调分布式应用的基本服务，它向外部应用暴露一组通用服务——分布式同步（Distributed Synchronization）、命名服务（Naming Service）、集群维护（Group Maintenance）等，简化分布式应用协调及其管理的难度，提供高性能的分布式服务。

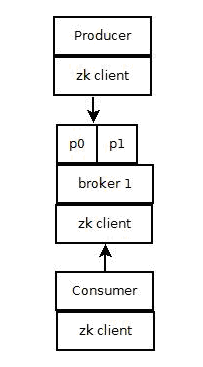
ZooKeeper本身可以以单机模式安装运行，不过它的长处在于通过分布式ZooKeeper集群（一个Leader，多个Follower），基于一定的策略来保证ZooKeeper集群的稳定性和可用性，从而实现分布式应用的可靠性。

本文着重介绍kafka，对zookeeper只做简要介绍。

## Zookeeper在kafka中的应用

Kafka使用zookeeper作为其分布式协调框架，很好的将消息生产、消息存储、消息消费的过程结合在一起。同时借助zookeeper，kafka能够生产者、消费者和broker在内的所以组件在无状态的情况下，建立起生产者和消费者的订阅关系，并实现生产者与消费者的负载均衡。

正如ZooKeeper用于分布式系统的协调和促进，Kafka使用ZooKeeper也是基于相同的原因。ZooKeeper用于管理、协调Kafka代理。每个Kafka代理都通过ZooKeeper协调其它Kafka代理。当Kafka系统中新增了代理或者某个代理故障失效时，ZooKeeper服务将通知生产者和消费者。生产者和消费者据此开始与其它代理协调工作。



1) Producer端使用zookeeper用来"发现"broker列表，以及和Topic下每个partition leader建立socket连接并发送消息。

2) Broker端使用zookeeper用来注册broker信息，已经监测partitionleader存活性。

3) Consumer端使用zookeeper用来注册consumer信息，其中包括consumer消费的partition列表等，同时也用来发现broker列表，并和partition leader建立socket连接，并获取消息。

## 安装部署zookeeper

### 安装zookeeper

#### 下载zookeeper安装包并解压

zookeeper-3.4.9.tar.gz放置于/soft/zookeeper下

[root@devhost15-5 zookeeper]# tar zxvf zookeeper-3.4.9.tar.gz

#### zookeeper目录介绍

bin目录：zk的可执行脚本目录，包括zk服务进程，zk客户端，等脚本。其中，.sh是Linux环境下的脚本，.cmd是Windows环境下的脚本。

conf目录：配置文件目录。zoo\_sample.cfg为样例配置文件，需要修改为自己的名称，一般为zoo.cfg。log4j.properties为日志配置文件。

Lib目录：zk依赖的包。

contrib目录：一些用于操作zk的工具包。

recipes目录 ：zk某些用法的代码示例

#### 修改配置

conf目录下提供了配置的样例zoo\_sample.cfg。要将zk运行起来，需要新建一个名称为zoo.cfg的配置文件。从zoo\_sample.cfg复制一个文件名为zoo.cfg后，修改配置如下。

[root@devhost15-5 conf]# cp zoo\_sample.cfg zoo.cfg

[root@devhost15-5 conf]# vi zoo.cfg

修改如下：

dataDir= /soft/zookeeper/zookeeper-3.4.9/data #用于配置存储快照文件的目录。如果没有配置dataLogDir，那么事务日志也会存储在此目录。

clientPort=2181 #监听client连接的端口号

### 启动zookeeper

#### 启动命令

进入bin目录，执行命令

[root@devhost15-5 bin]# ./zkServer.sh start

ZooKeeper JMX enabled by default

Using config: /soft/zookeeper/zookeeper-3.4.9/bin/../conf/zoo.cfg

Starting zookeeper ... STARTED

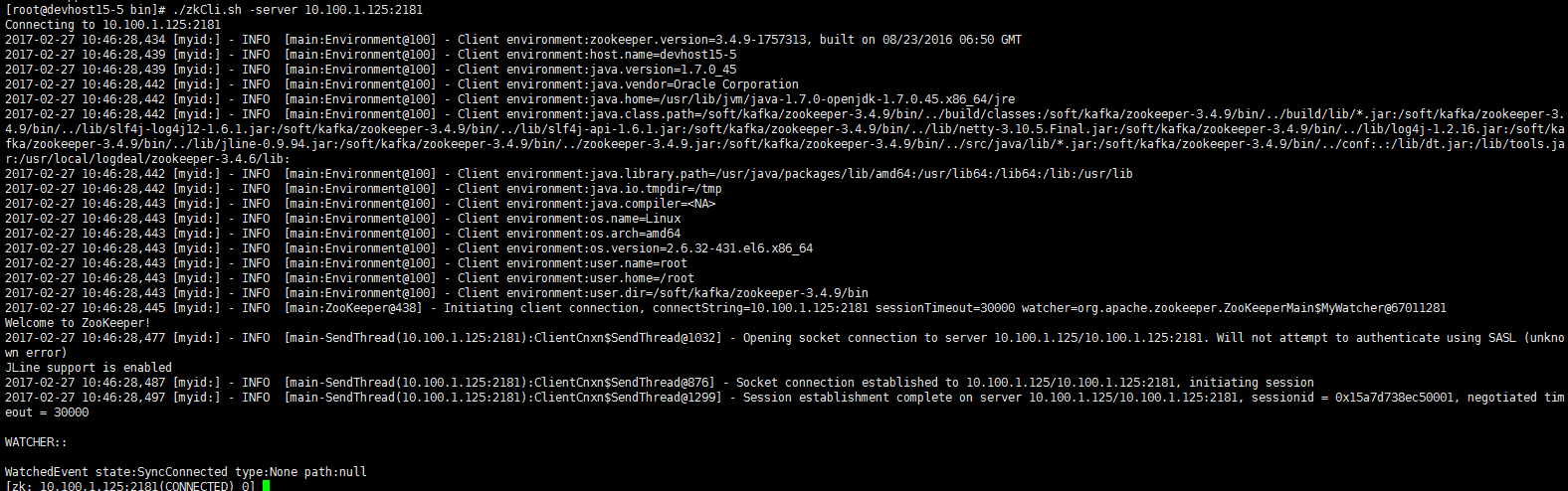
这个命令使得zk服务进程在后台进行。如果想在前台中运行以便查看服务器进程的输出日志，可以通过以下命令运行：

[root@devhost15-5 bin]#./zkServer.sh start-foreground

#### 连接查看

如果是连接同一台主机上的zk进程，那么直接运行bin/目录下的zkCli.sh，即可连接上zk。

[root@devhost15-5 bin]# ./zkCli.sh -server 10.100.1.125:2181



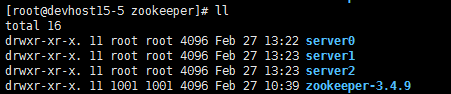
## Zookeeper集群配置【伪集群】

### 部署规模为3的单机伪机群

拷贝zookeeper目录，分别命名为server0、server1和server2：

[root@devhost15-5 zookeeper]# cp -R zookeeper-3.4.9 server0 server1 server2

拷贝完成后如下：



### 配置

#### 修改各server配置文件

* **server0**

[root@devhost15-5 conf]# vi /soft/zookeeper/server0/conf/zoo.cfg

dataDir=/soft/zookeeper/server0/data

clientPort=2190

server.0=10.100.1.125:8880:7770

server.1=10.100.1.125:8881:7771

server.2=10.100.1.125:8882:7772

* **server1**

[root@devhost15-5 conf]# vi /soft/zookeeper/server1/conf/zoo.cfg

dataDir=/soft/zookeeper/server1/data

clientPort=2191

server.0=10.100.1.125:8880:7770

server.1=10.100.1.125:8881:7771

server.2=10.100.1.125:8882:7772

* **server2**

[root@devhost15-5 conf]# vi /soft/zookeeper/server2/conf/zoo.cfg

dataDir=/soft/zookeeper/server2/data

clientPort=2192

server.0=10.100.1.125:8880:7770

server.1=10.100.1.125:8881:7771

server.2=10.100.1.125:8882:7772

注：

server.id=host:port1:port2

其中id为一个数字，表示zk进程的id，这个id也是dataDir目录下myid文件的内容。 host是该zk进程所在的IP地址，port1表示follower和leader交换消息所使用的端口，port2表示选举leader所使用的端口。

#### 创建myid文件

* **server0**

echo "0" > /soft/zookeeper/server0/data/myid

* **server1**

echo "1" > /soft/zookeeper/server1/data/myid

* **server2**

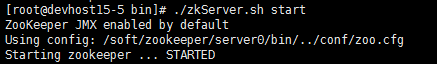
echo "2" > /soft/zookeeper/server2/data/myid

### 启动

#### 启动服务

分别进入三个服务器的/zookeeper/bin目录下，启动服务。

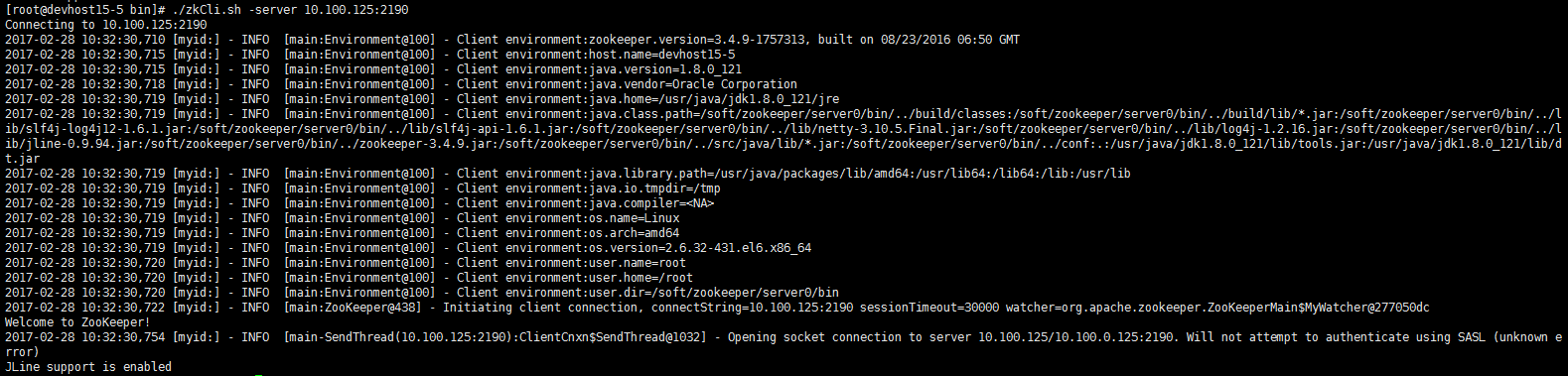
[root@devhost15-5 bin]# ./zkServer.sh start



#### 接入客户端

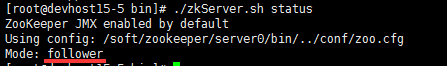
进入任意一个服务器的zookeeper/bin目录下，启动一个客户端，接入服务。

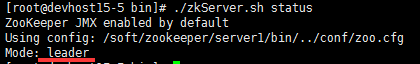
[root@devhost15-5 bin]# ./zkCli.sh -server 10.100.1.125:2190

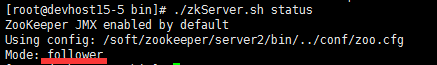


### 集群状态查询

[root@devhost15-5 bin]# ./zkServer.sh status







通过上面状态查询结果可见，server1是集群的Leader，其余的两个结点是Follower。

# 安装部署kafka

## 环境配置

JDK版本：1.8.0\_121

## 安装Kafka

### 下载Kafka并解压

kafka\_2.12-0.10.2.0.tgz放置于/soft/kafka目录下：

[root@devhost15-5 kafka]# pwd

/soft/kafka

解压缩：

[root@devhost15-5 kafka]# tar zxvf kafka\_2.12-0.10.2.0.tgz

### Kafka目录介绍

/bin 操作kafka的可执行脚本，还包含windows下脚本

/config 配置文件所在目录

/libs 依赖库目录

/logs 日志数据目录，目录kafka把server端日志分为5种类型，分为:server,request,state,log-cleaner,controller

### 修改server.properties配置文件

修改config下的server.properties文件

[root@devhost15-5 kafka]# cd kafka\_2.12-0.10.2.0/config/

[root@devhost15-5 config]# vi server.properties

修改如下：

broker.id=0 #每一个broker在集群中的唯一表示

log.dirs=/soft/kafka/kafka\_2.12-0.10.2.0/log # kafka数据的存放地址

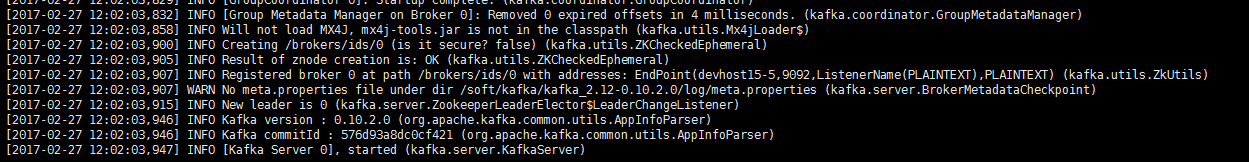
listeners=PLAINTEXT://10.100.1.125:9092 #监听器地址

zookeeper.connect=10.100.1.125:2181 # zookeeper的地址，集群用逗号隔开

## 启动kafka

### 启动命令

进入kafka目录，输入 bin/kafka-server-start.sh config/server.properties &



### 检测2181与9092端口

[root@devhost15-5 kafka\_2.12-0.10.2.0]# netstat -tunlp|egrep "(2181|9092)"



说明：

Kafka的进程ID为12727，占用端口为9092

QuorumPeerMain为对应的zookeeper实例，进程ID为8389，在2181端口监听

### 连通性测试

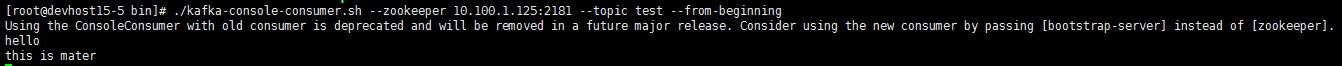
1. 启动2个XSHELL客户端，一个用于生产者发送消息，一个用于消费者接受消息。
2. 运行producer，随机敲入几个字符，相当于把这个敲入的字符消息发送给队列[bin目录下]

[root@devhost15-5 bin]# ./kafka-console-producer.sh --broker-list 10.100.1.125:9092 --topic test



1. 运行consumer，可以看到刚才发送的消息列表[bin目录下]

[root@devhost15-5 bin]# ./kafka-console-consumer.sh --zookeeper 10.100.1.125:2181 --topic test --from-beginning



注意：

producer，指定的Socket(10.100.1.125:9092),说明生产者的消息要发往kafka，即是broker

consumer, 指定的Socket(10.100.1.125:2181),说明消费者的消息来自zookeeper（协调转发）

# kafka集群配置【伪集群】

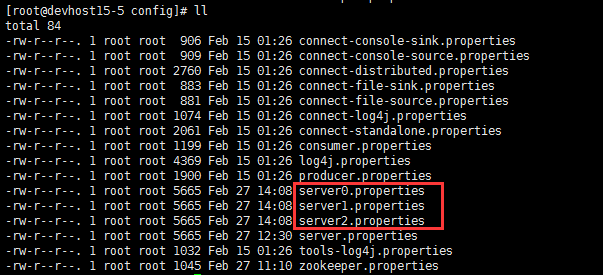
## 部署3个broker的伪集群

在kafka/config目录下拷贝server.properties，分别为server0.properties、server1.properties和server2.properties

[root@devhost15-5 config]# cp server.properties server0.properties

[root@devhost15-5 config]# cp server.properties server1.properties

[root@devhost15-5 config]# cp server.properties server2.properties



## 配置

修改各broker的配置文件

* **Broker0**

[root@devhost15-5 config]# vi server0.properties

broker.id=0

listeners=PLAINTEXT://10.100.1.125:9093

log.dirs=/soft/kafka/kafka\_2.12-0.10.2.0/log0

* **Broker1**

[root@devhost15-5 config]# vi server0.properties

broker.id=1

listeners=PLAINTEXT://10.100.1.125:9094

log.dirs=/soft/kafka/kafka\_2.12-0.10.2.0/log1

* **Broker2**

[root@devhost15-5 config]# vi server0.properties

broker.id=2

listeners=PLAINTEXT://10.100.1.125:9095

log.dirs=/soft/kafka/kafka\_2.12-0.10.2.0/log2

## 启动

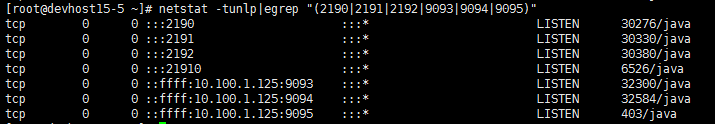
### 启动命令

[root@devhost15-5 kafka\_2.12-0.10.2.0]# bin/kafka-server-start.sh config/server0.properties & #启动broker0

[root@devhost15-5 kafka\_2.12-0.10.2.0]# bin/kafka-server-start.sh config/server1.properties & #启动broker1

[root@devhost15-5 kafka\_2.12-0.10.2.0]# bin/kafka-server-start.sh config/server2.properties & #启动broker2

### 查看端口

[root@devhost15-5 ~]# netstat -tunlp|egrep "(2190|2191|2192|9093|9094|9095)"

3个zookeeper在2190,2191,2192端口上监听，3个kafka cluster(broker)分别在端口9092,9093,9094监听。

## 验证

### 创建topic

[root@devhost15-5 kafka\_2.12-0.10.2.0]# bin/kafka-topics.sh --create --topic topic\_1 --partitions 1 --replication-factor 3 \--zookeeper 10.100.1.125:2190

[root@devhost15-5 kafka\_2.12-0.10.2.0]# bin/kafka-topics.sh --create --topic topic\_2 --partitions 2 --replication-factor 3 \--zookeeper 10.100.1.125:2190

[root@devhost15-5 kafka\_2.12-0.10.2.0]# bin/kafka-topics.sh --create --topic topic\_3 --partitions 3 --replication-factor 3 \--zookeeper 10.100.1.125:2190

注：--topic用来指定topic名称，--partitions指定分区数量，--replication-factor指定备份数量

### 查看topic创建情况

[root@devhost15-5 kafka\_2.12-0.10.2.0]# bin/kafka-topics.sh --list --zookeeper 10.100.1.125:2190

topic\_1

topic\_2

topic\_3

[root@devhost15-5 kafka\_2.12-0.10.2.0]# bin/kafka-topics.sh --describe --zookeeper 10.100.1.125:2190

Topic:topic\_1 PartitionCount:1 ReplicationFactor:3 Configs:

Topic: topic\_1 Partition: 0 Leader: 0 Replicas: 0,1,2 Isr: 0,1,2

Topic:topic\_2 PartitionCount:2 ReplicationFactor:3 Configs:

Topic: topic\_2 Partition: 0 Leader: 0 Replicas: 0,1,2 Isr: 0,1,2

Topic: topic\_2 Partition: 1 Leader: 1 Replicas: 1,2,0 Isr: 1,2,0

Topic:topic\_3 PartitionCount:3 ReplicationFactor:3 Configs:

Topic: topic\_3 Partition: 0 Leader: 1 Replicas: 1,2,0 Isr: 1,2,0

Topic: topic\_3 Partition: 1 Leader: 2 Replicas: 2,0,1 Isr: 2,0,1

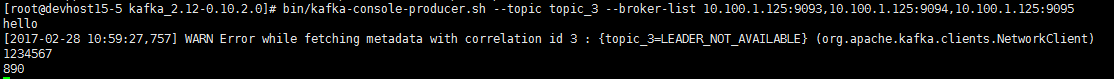
Topic: topic\_3 Partition: 2 Leader: 0 Replicas: 0,1,2 Isr: 0,1,2

### 模拟客户端发送，接受消息

* **发送消息**

用Kafka提供的生产者客户端启动一个生产者进程来发送消息

[root@devhost15-5 kafka\_2.12-0.10.2.0]# bin/kafka-console-producer.sh --topic topic --broker-list 10.100.1.125:9093,10.100.1.125:9094,10.100.1.125:9095



注：--topic定义了生产者发送给哪个topic ；--broker-list定义了生产者要推送消息的broker地址，以<IP地址:端口>形式。

* **接收消息**

[root@devhost15-5 kafka\_2.12-0.10.2.0]# bin/kafka-console-consumer.sh --topic topic --zookeeper 10.100.1.125:2190 --from-beginning

