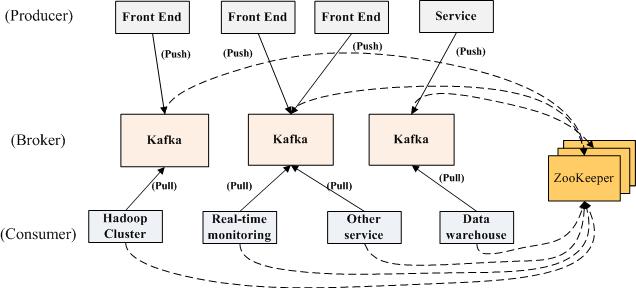
Kafka深度分析

# 架构

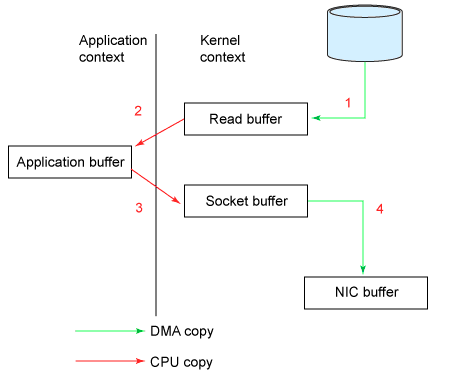


kafka是显式分布式架构，producer、broker（Kafka）和consumer都可以有多个。Kafka的运行依赖于ZooKeeper，Producer推送消息给kafka，Consumer从kafka拉消息。

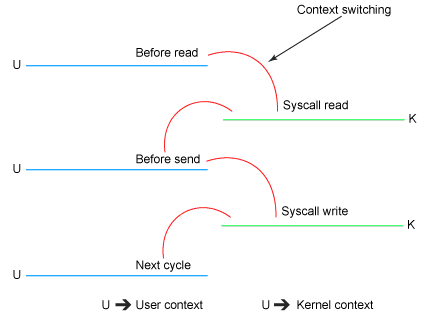
## kafka关键技术点

**（1）  zero-copy**

在Kafka上，有两个原因可能导致低效：1）太多的网络请求 2）过多的字节拷贝。为了提高效率，Kafka把message分成一组一组的，每次请求会把一组message发给相应的consumer。 此外， 为了减少字节拷贝，采用了sendfile系统调用。为了理解sendfile原理，先说一下传统的利用socket发送文件要进行拷贝：

[](http://dongxicheng.org/wp-content/uploads/2011/03/zero-copy02.gif)

Sendfile系统调用：

[](http://dongxicheng.org/wp-content/uploads/2011/03/zero-copy11.gif)

**（2） Exactly once message transfer**

怎样记录每个consumer处理的信息的状态？在Kafka中仅保存了每个consumer已经处理数据的offset。这样有两个好处：1）保存的数据量少 2）当consumer出错时，重新启动consumer处理数据时，只需从最近的offset开始处理数据即可。

**（3）Push/pull**

Producer 向Kafka（push）推数据，consumer 从kafka 拉（pull）数据。

**（4）负载均衡和容错**

Producer和broker之间没有负载均衡机制。  
broker和consumer之间利用zookeeper进行负载均衡。所有broker和consumer都会在zookeeper中进行注册，且zookeeper会保存他们的一些元数据信息。如果某个broker和consumer发生了变化，所有其他的broker和consumer都会得到通知。

# kafka术语

## Topic

Topic,是KAFKA对消息分类的依据;一条消息,必须有一个与之对应的Topic;

比如现在又两个Topic,分别是TopicA和TopicB,Producer向TopicA发送一个消息messageA,然后向TopicB发送一个消息messaeB;那么,订阅TopicA的Consumer就会收到消息messageA,订阅TopicB的Consumer就会收到消息messaeB;(每个Consumer可以同时订阅多个Topic,也即是说,同时订阅TopicA和TopicB的Consumer可以收到messageA和messaeB)。

同一个Group id的consumers在同一个Topic的同一条消息只能被一个consumer消费，实现了点对点模式，不同Group id的Consumers在同一个Topic上的同一条消息可以同时消费到，则实现了发布订阅模式。通过Consumer的Group id实现了JMS的消息模式

## Message

Message就是消息,是KAfKA操作的对象,消息是按照Topic存储的;

KAFKA中按照一定的期限保存着所有发布过的Message,不管这些Message是否被消费过;例如这些Message的保存期限被这只为两天,那么一条Message从发布开始的两天时间内是可用的,超过保存期限的消息会被清空以释放存储空间。

消息都是以字节数组进行网络传递。

## Partition

每一个Topic可以有多个Partition,这样做是为了提高KAFKA系统的并发能力，每个Partition中按照消息发送的顺序保存着Producer发来的消息,每个消息用ID标识,代表这个消息在改Partition中的偏移量,这样,知道了ID,就可以方便的定位一个消息了;每个新提交过来的消息,被追加到Partition的尾部;如果一个Partition被写满了,就不再追加;(注意,KAFKA不保证不同Partition之间的消息有序保存)

## Leader

Partition中负责消息读写的节点;Leader是从Partition的节点中随机选取的。每个Partition都会在集中的其中一台服务器存在Leader。一个Topic如果有多个Partition，则会有多个Leader。

## ReplicationFactor

一个Partition中复制数据的所有节点,包括已经挂了的;数量不会超过集群中broker的数量

## isr

ReplicationFactor的子集,存活的且和Leader保持同步的节点;

## Consumer Group

传统的消息系统提供两种使用方式:队列和发布-订阅;

队列：是一个池中有若干个Consumer,一条消息发出来以后,被其中的一个Consumer消费;

发布-订阅：是一个消息被广播出去,之后被所有订阅该主题的Consumer消费;

KAFKA提供的使用方式可以达到以上两种方式的效果:Consumer Group;

每一个Consumer用Consumer Group Name标识自己,当一条消息产生后,改消息被订阅了其Topic的Consumer Group收到,之后被这个Consumer Group中的一个Consumer消费;

如果所有的Consumer都在同一个Consumer Group中,那么这就和传统的队列形式的消息系统一样了;

如果每一个Consumer都在一个不同的Consumer Group中,那么就和传统的发布-订阅的形式一样了;

## Offset

消费者自己维护当前读取数据的offser，或者同步到zookeeper。*auto.commit.interval.ms* 是consumer同步offset到zookeeper的时间间隔。这个值设置问题会影响到多线程consumer，重复读取的问题。

# 安装启动配置环境

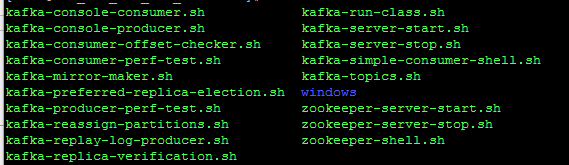
## 安装

下载kafka\_2.11-0.8.2.1，并在linux上解压

> **tar -xzf kafka\_2.11-0.8.2.1.tgz**

> **cd kafka\_2.11-0.8.2.1/bin**

可用的命令如下:



## 启动命令

Kafka需要用到zookeeper,所有首先需要启动zookeeper。

> **./zookeeper-server-start.sh ../config/zookeeper.properties &**

然后启动kafka服务

> **./kafka-server-start.sh ../config/server.properties &**

## 创建Topic

创建一个名字是”p2p”的topic，使用一个单独的partition和和一个replica

> **./kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic p2p**

使用命令查看topic

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

p2p

除了使用命令创建Topic外，可以让kafka自动创建，在客户端使用的时候，指定一个不存在的topic，kafka会自动给创建topic，自动创建将不能自定义partition和relica。

## 集群多broker

将上述的单节点kafka扩展为3个节点的集群。

从原始配置文件拷贝配置文件。

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

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

修改配置文件。

*config/server-1.properties:*

*broker.id=1*

*port=9093*

*log.dir=/tmp/kafka-logs-1*

*config/server-2.properties:*

*broker.id=2*

*port=9094*

*log.dir=/tmp/kafka-logs-2*

注意在集群中*broker.id*是唯一的。

现在在前面单一节点和zookeeper的基础上，再启动两个kafka节点。

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

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

创建一个新的topic，带三个ReplicationFactor

> **./kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 3 --partitions 1 --topic p2p-replicated-topic**

查看刚刚创建的topic。

> **./kafka-topics.sh --describe --zookeeper localhost:2181 --topic p2p-replicated-topic**

C:\Users\zhangwei01\AppData\Roaming\Tencent\Users\87064478\QQ\WinTemp\RichOle\V)0B(E%){{XC~PJR@XAK_XB.png

partiton： partion id，由于此处只有一个partition，因此partition id 为0  
leader：当前负责读写的lead broker id  
relicas：当前partition的所有replication broker  list  
isr：relicas的子集，只包含出于活动状态的broker

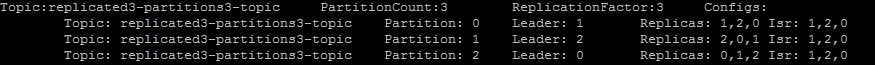
## Topic-Partition-Leader-ReplicationFactor 之间的关系样图

以上创建了三个节点的kafka集群，在集群上又用命令创建三个topic，分别是：

* replicated3-partitions3-topic：三份复制三个partition的topic
* replicated2-partitions3-topic：二份复制三个partition的topic
* test:1份复制，一个partition的topic

以我做测试创建的三个topic说明他们之间的关系。

>./kafka-topics.sh --describe --zookeeper localhost:2181 --topic replicated3-partitions3-topic



>./kafka-topics.sh --describe --zookeeper localhost:2181 --topic replicated2-partitions3-topic



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



以kafka当前的描述画出以下关系图：



从图上可以看到test没有备份，当broke Id 0 宕机后，虽然集群还有两个节点可以使用，但test这个topic却不能正常转发消息了。所以为了系统的可靠性，创建的replicas尽量的多，但却不能超过broker的数量。

# 客户端使用API

## Producer API

从0.8.2版本开始，apache提供了新的java版本的Producer的API。这个java版本在测试中表现比之前的scala客户端性能要好。Pom获取java客户端：

<dependency>

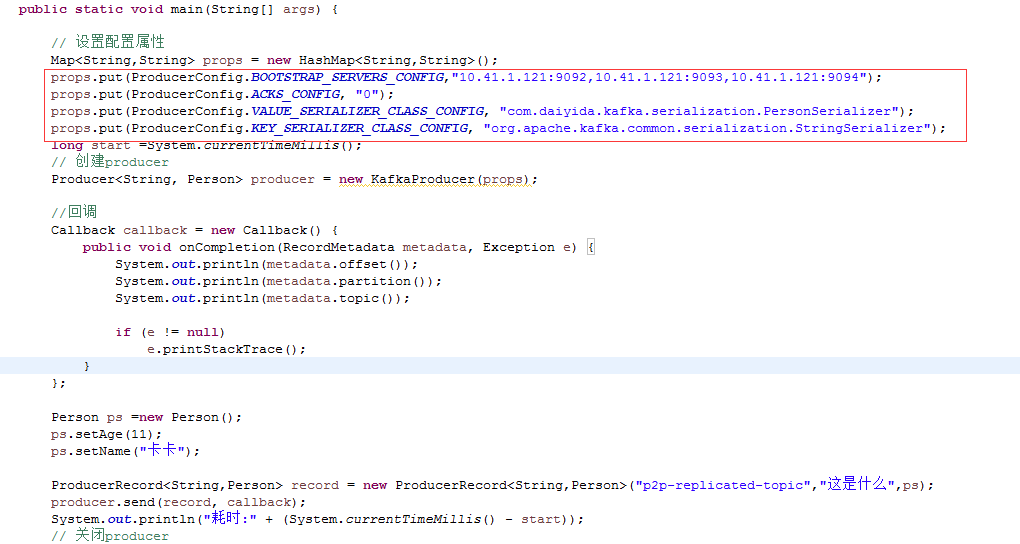
<groupId>org.apache.kafka</groupId>

<artifactId>kafka-clients</artifactId>

<version>0.8.2.1</version>

</dependency>

### Example



## Consumer API

Kafka 0.8.2.1版本已经放出了java版的consumer，看javadoc文档和代码不太匹配，也没有样例来说明java版的consumer的使用样例，这里还是用scala版的consumer API来使用。

Kafka提供了两套API给Consumer：

The high-level Consumer API：高度抽象的Consumer API，封装了很多consumer需要的高级功能，使用起来简单、方便

The SimpleConsumer API：只有最基本的链接、读取功能，可以自己去读offset，并指定offset的读取方式。适合于各种自定义

### High Level

class Consumer{

/\*\*

\* Create a ConsumerConnector:创建consumer connector

\*

\* @param config at the minimum, need to specify the groupid of the consumer and the zookeeper connection string zookeeper.connect.config参数作用：需要置顶consumer的groupid以及zookeeper连接字符串zookeeper.connect

\*/

public static kafka.javaapi.consumer.ConsumerConnector createJavaConsumerConnector(ConsumerConfig config);

}

/\*\*

\* V: type of the message： 消息类型

\* K: type of the optional key assciated with the message： 消息携带的可选关键字类型

\*/

public interface kafka.javaapi.consumer.ConsumerConnector {

/\*\*

\* Create a list of message streams of type T for each topic.：为每个topic创建T类型的消息流的列表

\*

\* @param topicCountMap a map of (topic, #streams) pair ： topic与streams的键值对

\* @param decoder a decoder that converts from Message to T ： 转换Message到T的解码器

\* @return a map of (topic, list of KafakStream) pairs. ： topic与KafkaStream列表的键值对

\* The number of items in the list is #streams . Each stream supports

\* an iterator over message/metadata pairs .：列表中项目的数量是#streams。每个stream都支持基于message/metadata 对的迭代器

\*/

public <K,V> Map<String, List<KafkaStream<K,V> > >

createMessageStreams( Map<String, Integer> topicCountMap, Decoder<K> keyDecoder, Decoder<V> valueDecoder);

/\*\*

\* Create a list of message streams of type T for each topic, using the default decoder.为每个topic创建T类型的消息列表。使用默认解码器

\*/

public Map<String, List<KafkaStream<byte[], byte[]>>> createMessageStreams(Map<String, Integer> topicCountMap);

/\*\*

   \*  Create a list of message streams for topics matching a wildcard.为匹配wildcard的topics创建消息流的列表

   \*

   \*  @param topicFilter a TopicFilter that specifies which topics to

   \*                    subscribe to (encapsulates a whitelist or a blacklist).指定将要订阅的topics的TopicFilter（封装了whitelist或者黑名单）

   \*  @param numStreams the number of message streams to return.将要返回的流的数量

   \*  @param keyDecoder a decoder that decodes the message key 可以解码关键字key的解码器

   \*  @param valueDecoder a decoder that decodes the message itself 可以解码消息本身的解码器

   \*  @return a list of KafkaStream. Each stream supports an

   \*          iterator over its MessageAndMetadata elements. 返回KafkaStream的列表。每个流都支持基于MessagesAndMetadata 元素的迭代器。

   \*/

 public <K,V> List<KafkaStream<K,V>>

    createMessageStreamsByFilter(TopicFilter topicFilter, int numStreams, Decoder<K> keyDecoder, Decoder<V> valueDecoder);

/\*\*

   \*  Create a list of message streams for topics matching a wildcard, using the default decoder.使用默认解码器，为匹配wildcard的topics创建消息流列表

   \*/

  public List<KafkaStream<byte[], byte[]>> createMessageStreamsByFilter(TopicFilter topicFilter, int numStreams);

  /\*\*

   \*  Create a list of message streams for topics matching a wildcard, using the default decoder, with one stream.使用默认解码器，为匹配wildcard的topics创建消息流列表

   \*/

  public List<KafkaStream<byte[], byte[]>> createMessageStreamsByFilter(TopicFilter topicFilter);

  /\*\*

   \*  Commit the offsets of all topic/partitions connected by this connector.通过connector提交所有topic/partitions的offsets

   \*/

  public void commitOffsets();

  /\*\*

   \*  Shut down the connector： 关闭connector

   \*/

  public void shutdown();

}

对大多数应用来说， high level已经足够了，一些应用要求的一些特征还没有出现high level consumer接口（例如，当重启consumer时，设置初始offset）。他们可以使用Simple Api。逻辑可能会有些复杂。

### Simple

使用Simple有以下缺点：

* 必须在程序中跟踪offset值
* 必须找出指定Topic Partition中的lead broker
* 必须处理broker的变动

class kafka.javaapi.consumer.SimpleConsumer {

  /\*\*

   \*  Fetch a set of messages from a topic.从topis抓取消息序列

   \*

   \*  @param request specifies the topic name, topic partition, starting byte offset, maximum bytes to be fetched.指定topic 名字，topic partition，开始的字节offset，抓取的最大字节数

   \*  @return a set of fetched messages

   \*/

  public FetchResponse fetch(kafka.javaapi.FetchRequest request);

  /\*\*

   \*  Fetch metadata for a sequence of topics.抓取一系列topics的metadata

   \*

   \*  @param request specifies the versionId, clientId, sequence of topics.指定versionId，clientId，topics

   \*  @return metadata for each topic in the request.返回此要求中每个topic的元素据

   \*/

  public kafka.javaapi.TopicMetadataResponse send(kafka.javaapi.TopicMetadataRequest request);

  /\*\*

   \*  Get a list of valid offsets (up to maxSize) before the given time.在给定的时间内返回正确偏移的列表

   \*

   \*  @param request a [[kafka.javaapi.OffsetRequest]] object.

   \*  @return a [[kafka.javaapi.OffsetResponse]] object.

   \*/

  public kafak.javaapi.OffsetResponse getOffsetsBefore(OffsetRequest request);

  /\*\*

   \* Close the SimpleConsumer.关闭

   \*/

  public void close();

}

## 配置

### Broker Config

核心关键配置：broker.id、log.dirs、zookeeper.connect

|  |  |  |
| --- | --- | --- |
| 参数 | 默认值 | 说明(解释) |
| broker.id |  | 每一个broker在集群中的唯一表示，非负正数。当该服务器的IP地址发生改变时，broker.id没有变化，则不会影响consumers的消息情况 |
| log.dirs | /tmp/kafka-logs | kafka数据的存放地址，多个地址的话用逗号分割/data/kafka-logs-1，/data/kafka-logs-2 |
| port | 9092 | broker server服务端口 |
| message.max.bytes | 1000000 | 表示消息体的最大大小，单位是字节 |
| num.network.threads | 3 | broker处理消息的最大线程数，一般情况下不需要去修改 |
| num.io.threads | 8 | broker处理磁盘IO的线程数，数值应该大于你的硬盘数 |
| background.threads | 10 | 一些后台任务处理的线程数，例如过期消息文件的删除等，一般情况下不需要去做修改 |
| queued.max.requests | 500 | 等待IO线程处理的请求队列最大数，若是等待IO的请求超过这个数值，那么会停止接受外部消息，应该是一种自我保护机制。 |
| host.name | null | broker的主机地址，若是设置了，那么会绑定到这个地址上，若是没有，会绑定到所有的接口上，并将其中之一发送到ZK，一般不设置 |
| advertised.host.name | null | If this is set this is the hostname that will be given out to producers, consumers, and other brokers to connect to. |
| advertised.port | null | The port to give out to producers, consumers, and other brokers to use in establishing connections. This only needs to be set if this port is different from the port the server should bind to. |
| socket.send.buffer.bytes | 100 \* 1024 | socket的发送缓冲区，socket的调优参数SO\_SNDBUFF |
| socket.receive.buffer.bytes | 100 \* 1024 | socket的接受缓冲区，socket的调优参数SO\_RCVBUFF |
| socket.request.max.bytes | 100 \* 1024 \* 1024 | socket请求的最大数值，防止内存溢出，必须小于Java heap size. |
| log.segment.bytes | 1024 \* 1024 \* 1024 | topic的分区是以一堆segment文件存储的，这个控制每个segment的大小，会被topic创建时的指定参数覆盖 |
| log.roll.hours | 24 \* 7 hours | 这个参数会在日志segment没有达到log.segment.bytes设置的大小，也会强制新建一个segment会被 topic创建时的指定参数覆盖 |
| log.cleanup.policy | delete | 日志清理策略选择有：delete和compact主要针对过期数据的处理，或是日志文件达到限制的额度，会被 topic创建时的指定参数覆盖 |
| log.retention.minutes | 7 days | 数据存储的最大时间超过这个时间会根据log.cleanup.policy设置的策略处理数据，也就是消费端能够多久去消费数据  log.retention.bytes和log.retention.minutes任意一个达到要求，都会执行删除，会被topic创建时的指定参数覆盖 |
| log.retention.bytes=-1 | -1 | topic每个分区的最大文件大小，一个topic的大小限制 =分区数\*log.retention.bytes。-1没有大小限log.retention.bytes和log.retention.minutes任意一个达到要求，都会执行删除，会被topic创建时的指定参数覆盖 |
| log.retention.check.interval.ms | 5 minutes | 文件大小检查的周期时间，是否处罚 log.cleanup.policy中设置的策略 |
| log.cleaner.enable | false | 是否开启日志压缩 |
| log.cleaner.threads | 1 | 日志压缩运行的线程数 |
| log.cleaner.io.max.bytes.per.second | Double.MaxValue | 日志压缩时候处理的最大大小 |
| log.cleaner.dedupe.buffer.size | 500\*1024\*1024 | 日志压缩去重时候的缓存空间，在空间允许的情况下，越大越好 |
| log.cleaner.io.buffer.size | 512\*1024 | 日志清理时候用到的IO块大小一般不需要修改 |
| log.cleaner.io.buffer.load.factor | 0.9 | 日志清理中hash表的扩大因子一般不需要修改 |
| log.cleaner.backoff.ms | 15000 | 检查是否清理日志清理的间隔 |
| log.cleaner.min.cleanable.ratio | 0.5 | 日志清理的频率控制，越大意味着更高效的清理，同时会存在一些空间上的浪费，会被topic创建时的指定参数覆盖 |
| log.cleaner.delete.retention.ms | 1 day | 对于压缩的日志保留的最长时间，也是客户端消费消息的最长时间，同log.retention.minutes的区别在于一个控制未压缩数据，一个控制压缩后的数据。会被topic创建时的指定参数覆盖 |
| log.index.size.max.bytes | 10 \* 1024 \* 1024 | 对于segment日志的索引文件大小限制，会被topic创建时的指定参数覆盖 |
| log.index.interval.bytes | 4096 | 当执行一个fetch操作后，需要一定的空间来扫描最近的offset大小，设置越大，代表扫描速度越快，但是也更好内存，一般情况下不需要搭理这个参数 |
| log.flush.interval.messages | Long.MaxValue | log文件”sync”到磁盘之前累积的消息条数,因为磁盘IO操作是一个慢操作,但又是一个”数据可靠性"的必要手段,所以此参数的设置,需要在**"数据可靠性"**与"性能"之间做必要的权衡.如果此值过大,将会导致每次"fsync"的时间较长(IO阻塞),如果此值过小,将会导致**"fsync"**的次数较多,这也意味着整体的client请求有一定的延迟.物理server故障,将会导致没有fsync的消息丢失. |
| log.flush.scheduler.interval.ms | Long.MaxValue | 检查是否需要固化到硬盘的时间间隔 |
| log.flush.interval.ms = None | Long.MaxValue | 仅仅通过interval来控制消息的磁盘写入时机,是不足的.此参数用于控制**"fsync"**的时间间隔,如果消息量始终没有达到阀值,但是离上一次磁盘同步的时间间隔达到阀值,也将触发. |
| log.delete.delay.ms | 60000 | 文件在索引中清除后保留的时间一般不需要去修改 |
| log.flush.offset.checkpoint.interval.ms | 60000 | 控制上次固化硬盘的时间点，以便于数据恢复一般不需要去修改 |
| log.segment.delete.delay.ms | 60000 | the amount of time to wait before deleting a file from the filesystem. |
| auto.create.topics.enable | true | 是否允许自动创建topic，若是**false**，就需要通过命令创建topic |
| **default**.replication.factor | 1 | 自动创建的topic默认 replication factor |
| num.partitions | 1 | 每个topic的分区个数，若是在topic创建时候没有指定的话会被topic创建时的指定参数覆盖 |
| 以下是kafka中Leader,replicas配置参数 |  |  |
| controller.socket.timeout.ms | 30000 | partition leader与replicas之间通讯时,socket的超时时间 |
| controller.message.queue.size | Int.MaxValue | partition leader与replicas数据同步时,消息的队列尺寸 |
| replica.lag.time.max.ms | 10000 | replicas响应partition leader的最长等待时间，若是超过这个时间，就将replicas列入ISR(in-sync replicas)，并认为它是死的，不会再加入管理中 |
| replica.lag.max.messages | 4000 | 如果follower落后与leader太多,将会认为此follower[或者说partition relicas]已经失效  ##通常,在follower与leader通讯时,因为网络延迟或者链接断开,总会导致replicas中消息同步滞后  ##如果消息之后太多,leader将认为此follower网络延迟较大或者消息吞吐能力有限,将会把此replicas迁移  ##到其他follower中.  ##在broker数量较少,或者网络不足的环境中,建议提高此值. |
| replica.socket.timeout.ms | 30 \* 1000 | follower与leader之间的socket超时时间 |
| replica.socket.receive.buffer.bytes | 64 \* 1024 | leader复制时候的socket缓存大小 |
| replica.fetch.max.bytes | 1024\*1024 | replicas每次获取数据的最大大小 |
| replica.fetch.wait.max.ms | 500 | replicas同leader之间通信的最大等待时间，失败了会重试 |
| replica.fetch.min.bytes | 1 | fetch的最小数据尺寸,如果leader中尚未同步的数据不足此值,将会阻塞,直到满足条件 |
| num.replica.fetchers | 1 | leader进行复制的线程数，增大这个数值会增加follower的IO |
| replica.high.watermark.checkpoint.interval.ms | 5000 | 每个replica检查是否将最高水位进行固化的频率 |
| fetch.purgatory.purge.interval.requests | 1000 | The purge interval (in number of requests) of the fetch request purgatory. |
| producer.purgatory.purge.interval.requests | 6000 | The purge interval (in number of requests) of the producer request purgatory. |
| controlled.shutdown.enable | true | 是否允许控制器关闭broker ,若是设置为**true**,会关闭所有在这个broker上的leader，并转移到其他broker |
| controlled.shutdown.max.retries | 3 | 控制器关闭的尝试次数 |
| controlled.shutdown.retry.backoff.ms | 5000 | 每次关闭尝试的时间间隔 |
| auto.leader.rebalance.enable | ture | 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 | leader的不平衡比例，若是超过这个数值，会对分区进行重新的平衡 |
| leader.imbalance.check.interval.seconds | 300 | 检查leader是否不平衡的时间间隔 |
| offset.metadata.max.bytes | 4096 | 客户端保留offset信息的最大空间大小 |
| zookeeper.connect | null | zookeeper集群的地址，可以是多个，多个之间用逗号分割hostname1:port1,hostname2:port2,hostname3:port3 |
| zookeeper.session.timeout.ms | 6000 | ZooKeeper的最大超时时间，就是心跳的间隔，若是没有反映，那么认为已经死了，不易过大 |
| zookeeper.connection.timeout.ms | 6000 | ZooKeeper的连接超时时间 |
| zookeeper.sync.time.ms | 2000 | ZooKeeper集群中leader和follower之间的同步实际那 |
| max.connections.per.ip | Int.MaxValue | The maximum number of connections that a broker allows from each ip address. |
| max.connections.per.ip.overrides |  | Per-ip or hostname overrides to the default maximum number of connections. |
| connections.max.idle.ms | 600000 | Idle connections timeout: the server socket processor threads close the connections that idle more than this. |
| log.roll.jitter.{ms,hours} | 0 | 随机的一个抖动时间 |
| 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 | Enable delete topic. |
| 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 |
| offsets.topic.retention.minutes | 1440 | 存在时间超过这个时间限制的offsets都将被标记为待删除 |
| offsets.retention.check.interval.ms | 600000 | offset管理器检查陈旧offsets的频率 |
| offsets.topic.replication.factor | 3 | topic的offset的备份份数。建议设置更高的数字保证更高的可用性 |
| offsets.topic.segment.bytes | 104857600 | offsets topic的segment大小 |
| offsets.load.buffer.size | 5242880 | 这项设置与批量尺寸相关，当从offsets segment中读取时使用。 |
| offsets.commit.required.acks | -1 | 在offset  commit可以接受之前，需要设置确认的数目，一般不需要更改. 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. |

topic-level的配置

有关topics的配置既有全局的又有每个topic独有的配置。如果没有给定特定topic设置，则应用默认的全局设置。这些覆盖会在每次创建topic发生。下面的例子：创建一个topic，命名为my-topic，自定义最大消息尺寸以及刷新比率为：

>    bin/kafka-topics.sh --zookeeper localhost:2181 --create --topic my-topic --partitions 1   
        --replication-factor 1 --config max.message.bytes=64000 --config flush.messages=1

需要删除重写时，可以按照以下来做：

> bin/kafka-topics.sh --zookeeper localhost:2181 --alter --topic my-topic   
    --deleteConfig max.message.bytes

以下是topic-level的配置选项。server的默认配置在Server  Default Property列下给出了，设定这些默认值不会改变原有的设置

|  |  |  |  |
| --- | --- | --- | --- |
| Property | Default | Server Default Property | Description |
| cleanup.policy | delete | log.cleanup.policy | 要么是”delete“要么是”compact“； 这个字符串指明了针对旧日志部分的利用方式；默认方式（"delete"）将会丢弃旧的部分当他们的回收时间或者尺寸限制到达时。”compact“将会进行日志压缩 |
| delete.retention.ms | 86400000 (24 hours) | log.cleaner.delete.retention.ms | 对于压缩日志保留的最长时间，也是客户端消费消息的最长时间，通log.retention.minutes的区别在于一个控制未压缩数据，一个控制压缩后的数据。此项配置可以在topic创建时的置顶参数覆盖 |
| flush.messages | None | log.flush.interval.messages | 此项配置指定时间间隔：强制进行fsync日志。例如，如果这个选项设置为1，那么每条消息之后都需要进行fsync，如果设置为5，则每5条消息就需要进行一次fsync。一般来说，建议你不要设置这个值。此参数的设置,需要在"数据可靠性"与"性能"之间做必要的权衡.如果此值过大,将会导致每次"fsync"的时间较长(IO阻塞),如果此值过小,将会导致"fsync"的次数较多,这也意味着整体的client请求有一定的延迟.物理server故障,将会导致没有fsync的消息丢失. |
| flush.ms | None | log.flush.interval.ms | 此项配置用来置顶强制进行fsync日志到磁盘的时间间隔；例如，如果设置为1000，那么每1000ms就需要进行一次fsync。一般不建议使用这个选项 |
| index.interval.bytes | 4096 | log.index.interval.bytes | 默认设置保证了我们每4096个字节就对消息添加一个索引，更多的索引使得阅读的消息更加靠近，但是索引规模却会由此增大；一般不需要改变这个选项 |
| max.message.bytes | 1000000 | max.message.bytes | kafka追加消息的最大尺寸。注意如果你增大这个尺寸，你也必须增大你consumer的fetch 尺寸，这样consumer才能fetch到这些最大尺寸的消息。 |
| min.cleanable.dirty.ratio | 0.5 | min.cleanable.dirty.ratio | 此项配置控制log压缩器试图进行清除日志的频率。默认情况下，将避免清除压缩率超过50%的日志。这个比率避免了最大的空间浪费 |
| min.insync.replicas | 1 | min.insync.replicas | 当producer设置request.required.acks为-1时，min.insync.replicas指定replicas的最小数目（必须确认每一个repica的写数据都是成功的），如果这个数目没有达到，producer会产生异常。 |
| retention.bytes | None | log.retention.bytes | 如果使用“delete”的retention  策略，这项配置就是指在删除日志之前，日志所能达到的最大尺寸。默认情况下，没有尺寸限制而只有时间限制 |
| retention.ms | 7 days | log.retention.minutes | 如果使用“delete”的retention策略，这项配置就是指删除日志前日志保存的时间。 |
| segment.bytes | 1GB | log.segment.bytes | kafka中log日志是分成一块块存储的，此配置是指log日志划分成块的大小 |
| segment.index.bytes | 10MB | log.index.size.max.bytes | 此配置是有关offsets和文件位置之间映射的索引文件的大小；一般不需要修改这个配置 |
| segment.ms | 7 days | log.roll.hours | 即使log的分块文件没有达到需要删除、压缩的大小，一旦log 的时间达到这个上限，就会强制新建一个log分块文件 |
| segment.jitter.ms | 0 | log.roll.jitter.{ms,hours} | The maximum jitter to subtract from logRollTimeMillis. |

### Producer API Config

这里只介绍使用的java API用的配置

核心配置：boostrap.servers、acks、value.serializer、key.serializer

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Type | Default | Importance | Description |
| boostrap.servers | list |  | high | 用于建立与kafka集群连接的host/port组。数据将会在所有servers上均衡加载，不管哪些server是指定用于bootstrapping。这个列表仅仅影响初始化的hosts（用于发现全部的servers）。这个列表格式： host1:port1,host2:port2,... 因为这些server仅仅是用于初始化的连接，以发现集群所有成员关系（可能会动态的变化），这个列表不需要包含所有的servers（你可能想要不止一个server，尽管这样，可能某个server宕机了）。如果没有server在这个列表出现，则发送数据会一直失败，直到列表可用。 |
| acks | string | 1 | high | producer需要server接收到数据之后发出的确认接收的信号，此项配置就是指procuder需要多少个这样的确认信号。此配置实际上代表了数据备份的可用性。以下设置为常用选项： （1）acks=0： 设置为0表示producer不需要等待任何确认收到的信息。副本将立即加到socket  buffer并认为已经发送。没有任何保障可以保证此种情况下server已经成功接收数据，同时重试配置不会发生作用（因为客户端不知道是否失败）回馈的offset会总是设置为-1； （2）acks=1： 这意味着至少要等待leader已经成功将数据写入本地log，但是并没有等待所有follower是否成功写入。这种情况下，如果follower没有成功备份数据，而此时leader又挂掉，则消息会丢失。 （3）acks=all： 这意味着leader需要等待所有备份都成功写入日志，这种策略会保证只要有一个备份存活就不会丢失数据。这是最强的保证。 （4）其他的设置，例如acks=2也是可以的，这将需要给定的acks数量，但是这种策略一般很少用。 |
| buffer.memory | long | 33554432 | high | producer可以用来缓存数据的内存大小。如果数据产生速度大于向broker发送的速度，producer会阻塞或者抛出异常，以“block.on.buffer.full”来表明。  这项设置将和producer能够使用的总内存相关，但并不是一个硬性的限制，因为不是producer使用的所有内存都是用于缓存。一些额外的内存会用于压缩（如果引入压缩机制），同样还有一些用于维护请求。 |
| compression.type | string | none | high | producer用于压缩数据的压缩类型。默认是无压缩。正确的选项值是none、gzip、snappy。 压缩最好用于批量处理，批量处理消息越多，压缩性能越好。 |
| retries | int | 0 | high | 设置大于0的值将使客户端重新发送任何数据，一旦这些数据发送失败。注意，这些重试与客户端接收到发送错误时的重试没有什么不同。允许重试将潜在的改变数据的顺序，如果这两个消息记录都是发送到同一个partition，则第一个消息失败第二个发送成功，则第二条消息会比第一条消息出现要早。 |
| batch.size | int | 16384 | medium | producer将试图批处理消息记录，以减少请求次数。这将改善client与server之间的性能。这项配置控制默认的批量处理消息字节数。 不会试图处理大于这个字节数的消息字节数。 发送到brokers的请求将包含多个批量处理，其中会包含对每个partition的一个请求。 较小的批量处理数值比较少用，并且可能降低吞吐量（0则会仅用批量处理）。较大的批量处理数值将会浪费更多内存空间，这样就需要分配特定批量处理数值的内存大小。 |
| client.id | string |  | medium | 当向server发出请求时，这个字符串会发送给server。目的是能够追踪请求源头，以此来允许ip/port许可列表之外的一些应用可以发送信息。这项应用可以设置任意字符串，因为没有任何功能性的目的，除了记录和跟踪 |
| linger.ms | long | 0 | medium | producer组将会汇总任何在请求与发送之间到达的消息记录一个单独批量的请求。通常来说，这只有在记录产生速度大于发送速度的时候才能发生。然而，在某些条件下，客户端将希望降低请求的数量，甚至降低到中等负载一下。这项设置将通过增加小的延迟来完成--即，不是立即发送一条记录，producer将会等待给定的延迟时间以允许其他消息记录发送，这些消息记录可以批量处理。这可以认为是TCP种Nagle的算法类似。这项设置设定了批量处理的更高的延迟边界：一旦我们获得某个partition的batch.size，他将会立即发送而不顾这项设置，然而如果我们获得消息字节数比这项设置要小的多，我们需要“linger”特定的时间以获取更多的消息。 这个设置默认为0，即没有延迟。设定linger.ms=5，例如，将会减少请求数目，但是同时会增加5ms的延迟。 |
| max.request.size | int | 1028576 | medium | 请求的最大字节数。这也是对最大记录尺寸的有效覆盖。注意：server具有自己对消息记录尺寸的覆盖，这些尺寸和这个设置不同。此项设置将会限制producer每次批量发送请求的数目，以防发出巨量的请求。 |
| receive.buffer.bytes | int | 32768 | medium | TCP receive缓存大小，当阅读数据时使用 |
| send.buffer.bytes | int | 131072 | medium | TCP send缓存大小，当发送数据时使用 |
| timeout.ms | int | 30000 | medium | 此配置选项控制server等待来自followers的确认的最大时间。如果确认的请求数目在此时间内没有实现，则会返回一个错误。这个超时限制是以server端度量的，没有包含请求的网络延迟 |
| block.on.buffer.full | boolean | true | low | 当我们内存缓存用尽时，必须停止接收新消息记录或者抛出错误。默认情况下，这个设置为真，然而某些阻塞可能不值得期待，因此立即抛出错误更好。设置为false则会这样：producer会抛出一个异常错误：BufferExhaustedException， 如果记录已经发送同时缓存已满 |
| metadata.fetch.timeout.ms | long | 60000 | low | 是指我们所获取的一些元素据的第一个时间数据。元素据包含：topic，host，partitions。此项配置是指当等待元素据fetch成功完成所需要的时间，否则会跑出异常给客户端。 |
| metadata.max.age.ms | long | 300000 | low | 以微秒为单位的时间，是在我们强制更新metadata的时间间隔。即使我们没有看到任何partition leadership改变。 |
| metric.reporters | list | [] | low | 类的列表，用于衡量指标。实现MetricReporter接口，将允许增加一些类，这些类在新的衡量指标产生时就会改变。JmxReporter总会包含用于注册JMX统计 |
| metrics.num.samples | int | 2 | low | 用于维护metrics的样本数 |
| metrics.sample.window.ms | long | 30000 | low | metrics系统维护可配置的样本数量，在一个可修正的window  size。这项配置配置了窗口大小，例如。我们可能在30s的期间维护两个样本。当一个窗口推出后，我们会擦除并重写最老的窗口 |
| recoonect.backoff.ms | long | 10 | low | 连接失败时，当我们重新连接时的等待时间。这避免了客户端反复重连 |
| retry.backoff.ms | long | 100 | low | 在试图重试失败的produce请求之前的等待时间。避免陷入发送-失败的死循环中。 |

### Consumer API Config

consumer基本配置如下：

       group.id

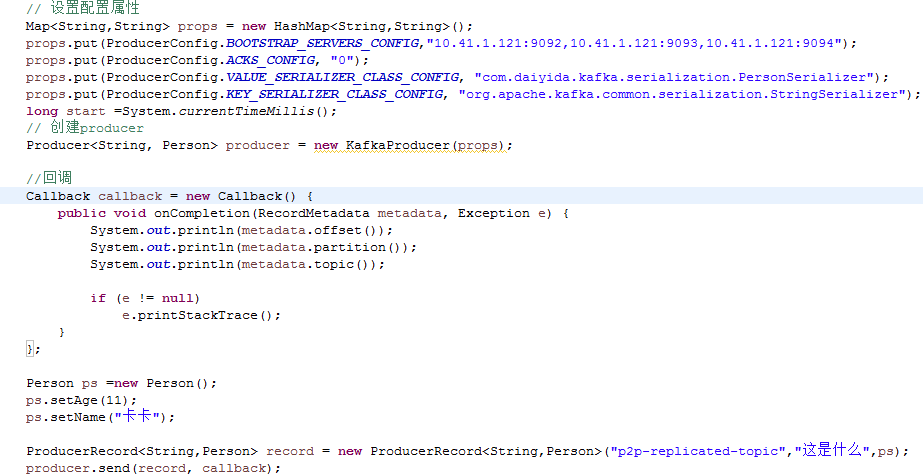
       zookeeper.connect

|  |  |  |
| --- | --- | --- |
| Property | Default | Description |
| group.id |  | 用来唯一标识consumer进程所在组的字符串，如果设置同样的group  id，表示这些processes都是属于同一个consumer  group |
| zookeeper.connect |  | 指定zookeeper的连接的字符串，格式是hostname：port，此处host和port都是zookeeper server的host和port，为避免某个zookeeper 机器宕机之后失联，你可以指定多个hostname：port，使用逗号作为分隔： hostname1：port1，hostname2：port2，hostname3：port3 可以在zookeeper连接字符串中加入zookeeper的chroot路径，此路径用于存放他自己的数据，方式： hostname1：port1，hostname2：port2，hostname3：port3/chroot/path |
| consumer.id | null | 不需要设置，一般自动产生 |
| socket.timeout.ms | 30\*100 | 网络请求的超时限制。真实的超时限制是   max.fetch.wait+socket.timeout.ms |
| socket.receive.buffer.bytes | 64\*1024 | socket用于接收网络请求的缓存大小 |
| fetch.message.max.bytes | 1024\*1024 | 每次fetch请求中，针对每次fetch消息的最大字节数。这些字节将会督导用于每个partition的内存中，因此，此设置将会控制consumer所使用的memory大小。这个fetch请求尺寸必须至少和server允许的最大消息尺寸相等，否则，producer可能发送的消息尺寸大于consumer所能消耗的尺寸。 |
| num.consumer.fetchers | 1 | 用于fetch数据的fetcher线程数 |
| auto.commit.enable | true | 如果为真，consumer所fetch的消息的offset将会自动的同步到zookeeper。这项提交的offset将在进程挂掉时，由新的consumer使用 |
| auto.commit.interval.ms | 60\*1000 | consumer向zookeeper提交offset的频率，单位是毫秒 |
| queued.max.message.chunks | 2 | 用于缓存消息的最大数目，以供consumption。每个chunk必须和fetch.message.max.bytes相同 |
| rebalance.max.retries | 4 | 当新的consumer加入到consumer  group时，consumers集合试图重新平衡分配到每个consumer的partitions数目。如果consumers集合改变了，当分配正在执行时，这个重新平衡会失败并重入 |
| fetch.min.bytes | 1 | 每次fetch请求时，server应该返回的最小字节数。如果没有足够的数据返回，请求会等待，直到足够的数据才会返回。 |
| fetch.wait.max.ms | 100 | 如果没有足够的数据能够满足fetch.min.bytes，则此项配置是指在应答fetch请求之前，server会阻塞的最大时间。 |
| rebalance.backoff.ms | 2000 | 在重试reblance之前backoff时间 |
| refresh.leader.backoff.ms | 200 | 在试图确定某个partition的leader是否失去他的leader地位之前，需要等待的backoff时间 |
| auto.offset.reset | largest | zookeeper中没有初始化的offset时，如果offset是以下值的回应： smallest：自动复位offset为smallest的offset largest：自动复位offset为largest的offset anything  else：向consumer抛出异常 |
| consumer.timeout.ms | -1 | 如果没有消息可用，即使等待特定的时间之后也没有，则抛出超时异常 |
| exclude.internal.topics | true | 是否将内部topics的消息暴露给consumer |
| paritition.assignment.strategy | range | 选择向consumer 流分配partitions的策略，可选值：range，roundrobin |
| client.id | group id value | 是用户特定的字符串，用来在每次请求中帮助跟踪调用。它应该可以逻辑上确认产生这个请求的应用 |
| zookeeper.session.timeout.ms | 6000 | zookeeper 会话的超时限制。如果consumer在这段时间内没有向zookeeper发送心跳信息，则它会被认为挂掉了，并且reblance将会产生 |
| zookeeper.connection.timeout.ms | 6000 | 客户端在建立通zookeeper连接中的最大等待时间 |
| zookeeper.sync.time.ms | 2000 | ZK follower可以落后ZK leader的最大时间 |
| offsets.storage | zookeeper | 用于存放offsets的地点： zookeeper或者kafka |
| offset.channel.backoff.ms | 1000 | 重新连接offsets channel或者是重试失败的offset的fetch/commit请求的backoff时间 |
| offsets.channel.socket.timeout.ms | 10000 | 当读取offset的fetch/commit请求回应的socket 超时限制。此超时限制是被consumerMetadata请求用来请求offset管理 |
| offsets.commit.max.retries | 5 | 重试offset commit的次数。这个重试只应用于offset  commits在shut-down之间。他 |
| dual.commit.enabled | true | 如果使用“kafka”作为offsets.storage，你可以二次提交offset到zookeeper(还有一次是提交到kafka）。在zookeeper-based的offset  storage到kafka-based的offset storage迁移时，这是必须的。对任意给定的consumer  group来说，比较安全的建议是当完成迁移之后就关闭这个选项 |
| partition.assignment.strategy | range | 在“range”和“roundrobin”策略之间选择一种作为分配partitions给consumer 数据流的策略； 循环的partition分配器分配所有可用的partitions以及所有可用consumer  线程。它会将partition循环的分配到consumer线程上。如果所有consumer实例的订阅都是确定的，则partitions的划分是确定的分布。循环分配策略只有在以下条件满足时才可以：（1）每个topic在每个consumer实力上都有同样数量的数据流。（2）订阅的topic的集合对于consumer  group中每个consumer实例来说都是确定的。 |

更多细节可以查看  scala类：  kafka.consumer.ConsumerConfig

## 样例代码

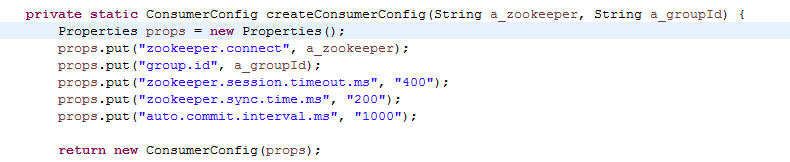
### Producer



具体参看附件

### Consumer

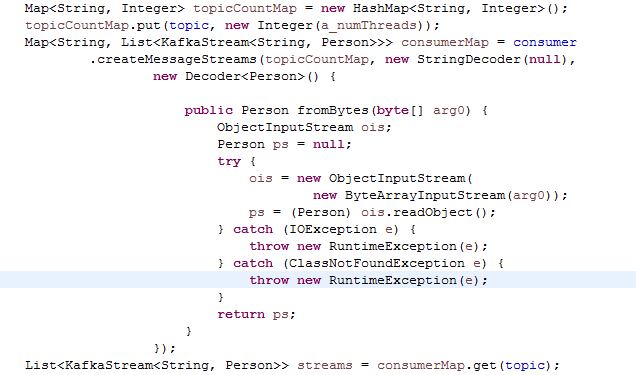
创建配置



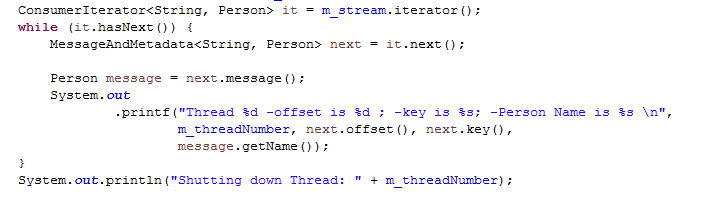
创建Consumer对象

C:\Users\zhangwei01\AppData\Roaming\Tencent\Users\87064478\QQ\WinTemp\RichOle\X(]3)OXMFZ{)7)IG5$6U839.png

创建KafakaStream对象



读取消息



具体参看附件

# 生产者均衡

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,有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,可以在producer端实现消息均衡分发.

异步发送：将多条消息暂且在客户端buffer起来，并将他们批量的发送到broker，小数据IO太多，会拖慢整体的网络延迟，批量延迟发送事实上提升了网络效率。不过这也有一定的隐患，比如说当producer失效时，那些尚未发送的消息将会丢失。

# 监控

yahoo为了简化开发者和服务工程师维护Kafka集群的工作，构建了一个叫做Kafka管理器的基于Web工具，叫做 Kafka Manager。这个管理工具可以很容易地发现分布在集群中的哪些topic分布不均匀，或者是分区在整个集群分布不均匀的的情况。

通过Kafka Manager用户能够更容易地发现集群中哪些主题或者分区分布不均匀，同时能够管理多个集群，能够更容易地检查集群的状态，能够创建主题，执行首选的副本选择，能够基于集群当前的状态生成分区分配，并基于生成的分配执行分区的重分配，此外，Kafka Manager还是一个非常好的可以快速查看集群状态的工具。

该软件是用Scala语言编写的。yahoo已经开源了Kafka Manager工具。这款Kafka集群管理工具主要支持以下几个功能：  
　　1、管理几个不同的集群；  
　　2、很容易地检查集群的状态(topics, brokers, 副本的分布, 分区的分布)；  
　　3、选择副本；  
　　4、产生分区分配(Generate partition assignments)基于集群的当前状态；  
　　5、重新分配分区。

6、支持kafka 0.8.2之上的版本删除Topic。

7、标记删除的分区的主题（0.8.2+）

8、批量产生分区分配为多个Topic

9、批量执行重新分配分区为个Topic

10、为已存在的Tpic增加分区。

11、选择是否启用JMX轮询代理和Topic的度量

## 编译安装kafka Manager的过程

### 安装sbt

sbt是scala的打包构建工具。编译kafka Manager之前需要安装sbt，下载地址：<http://www.scala-sbt.org/download.html>

我本地用的是windows版，执行安装文件就可以。

### 下载kafka Manager编译

kafka Manager是从github下载下来的，windows下先安装个git环境，git的环境安装这里就不介绍了。

在git的bash下，执行以下命令：

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

kafka Manager将会拷贝到本地目录内。

在windows的命令行下，进入kafka-manager的主目录：

cd kafka-manager

执行以下命令：

sbt clean dist

sbt在编译的时候会通过ivy下载很多依赖包，网络不好的情况下往往会很长时间，而且很有可能下载不成功。我本地就一直没有下载成功过，通过在国外vps上执行sbt，将依赖包下载到vps上，然后再打包ftp下载到本地，覆盖本地的ivy cache，通过这种方式实现kafka manager的完成编译。

### 运行kafka Manager

Kafka Manager编译打包后的，生成的打包文件在target\universal目录下，目录下有个zip文件，这个文件就是kafka Manager编译打包后的可运行包。将zip文件拷贝到linux下。执行uzip，进行解压缩。进入kafka Manager目录，修改conf目录下的application.conf，修改值kafka-manager.zkhosts="kafka-manager-zookeeper:2181"为真是的zookeeper的地址。保存

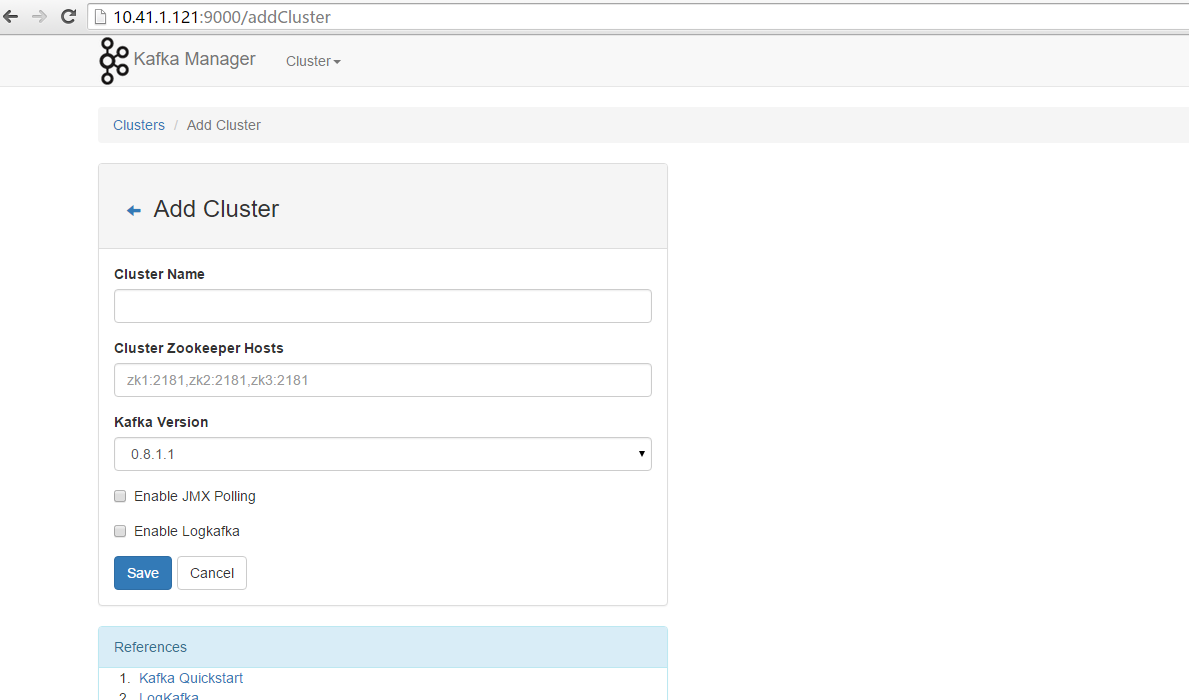
进入bin目录下执行命令：

../kafka-manager -Dconfig.file=../conf/application.conf

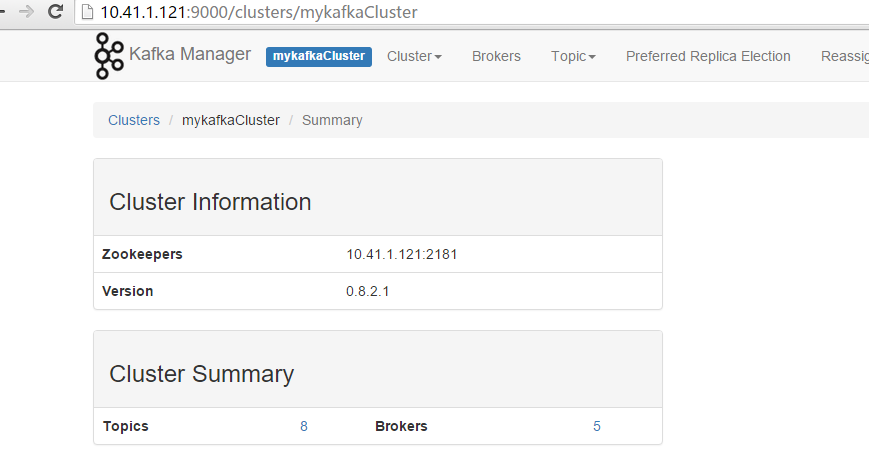
如果命令不能执行，请先给文件授权：

.chmod 777 kafka-manager

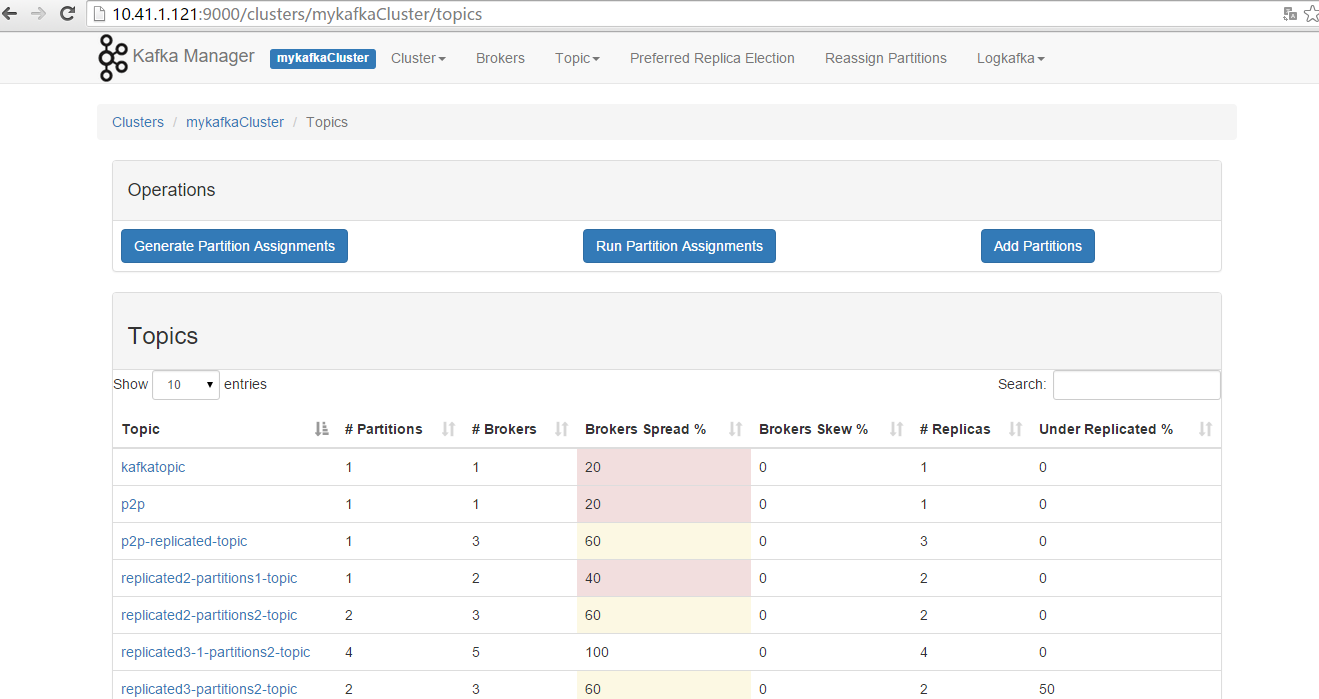
打开浏览器输入<http://x.x.x.x:9000>，即可进入kafka manager的管理页面。



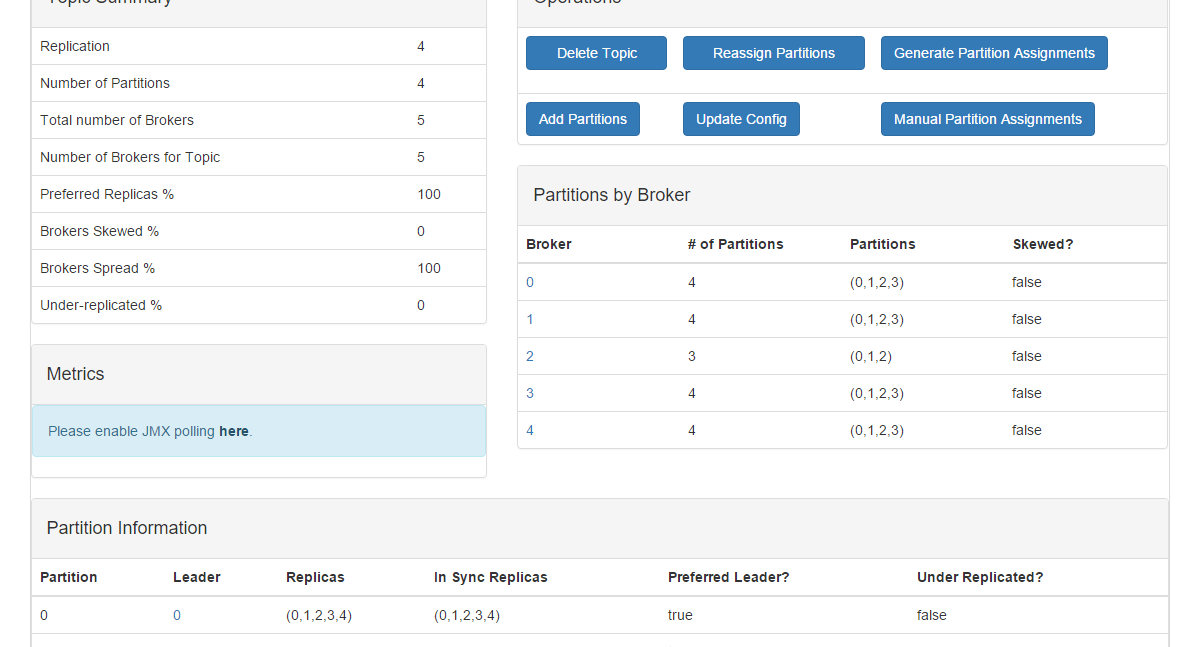
填写集群名字，zk的地址和端口，选择kafka的版本，我测试用的版本是0.8.2.1。然后保存。



这里可以看到集群的Topic数量和Brokers的数据。点击数据可以看详细信息。



在Topic列表，可以点击某个topic，查看详细。



用kafka Manager可以用很多维护kafka的操作，如删除Topic、增加Partition、重新分配Partition、Jmx监控等。

# 扩展集群

将新的kafka 服务器假如到集群中是相当简单的。但是这些新加入的服务器不会自动分配任何数据partition，除非现有的partition迁移到新的服务器上，否则新加入的服务器是不起作用的，直到新的Topic被创建。所以通常当你增加新的服务器到集群中的时候，你需要迁移存在的数据到新的服务器上。

迁移数据是一个手工初始化全程自动化的过程，kafka将会在新的服务器上创建一个follwer，并且允许完全复制partition上存在的数据。当新的服务器完全复制了partion的内容并加入到同步复制，之前存在的replicas将会删除他们的数据。

Partition重新分配工具能够跨broker迁移。在0.8.1版本，重新分配工具还不能自动获得分布且计算出如何移动。因此管理员需要手工指出那个topic那个partition应该如何移动。

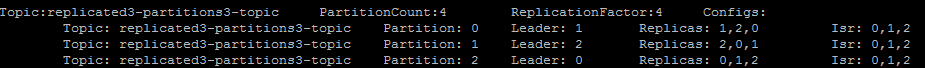
Partition重新分配工具有三种运行模式：

* 1. --generate: 对指定的topics移动所有的partition到新的brokers
  2. --execute: 指定 --reassignment-json-file用自定义的重新分配方案
  3. --verify: 确认最后一个-execute的所有Partition的重新分配状态

## Partition和Relicas扩展

官方网站上说kafka只支持partitions的增加，不支持Relicas的修改，但我通过下面的操作实现了即扩展partition，又实现了Relicas，需要进一步验证。

以对replicated3-partitions3-topic操作为例，对集群增加一个broker为3的服务器，则集群中有四个broker了,原始replicated3-partitions3-topic状态

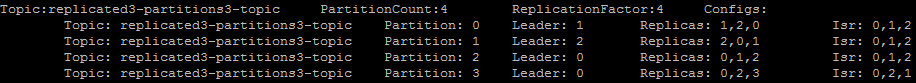


下面的步骤对replicated3-partitions3-topic进行扩展：

1. 扩展Partition

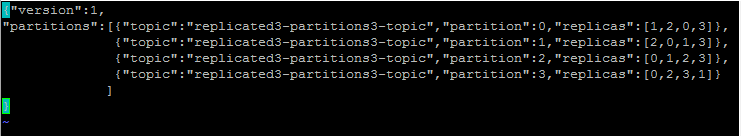
> ../kafka-topics.sh --zookeeper 10.41.1.121:2181 --alter --topic replicated3-partitions3-topic –partitions 4

执行命令后，查看状态如下



1. 使用重新分配工具，创建重新分配json文件

> Vim topics-to-move.json

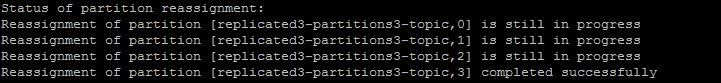


1. 执行重新分配

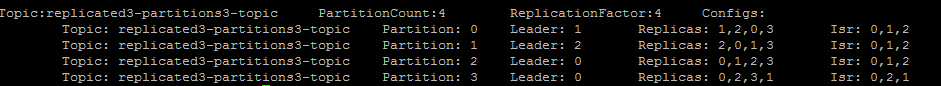
> ./kafka-reassign-partitions.sh --zookeeper 10.41.1.121:2181 --reassignment-json-file expand- topics-to-move.json --execute

4、查看执行分配的结果

> ../kafka-reassign-partitions.sh --zookeeper 10.41.1.121:2181 --reassignment-json-file topics-to-mov.json –verify

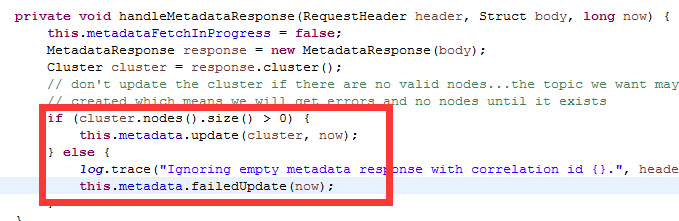


1. 再次查看replicated3-partitions3-topic的状态



## Producer平滑扩展

Producer在启动的时候，会手工配置初始化连接，通过boostrap.servers参数，集群扩展后，增加新的broker，Producer是怎么扩展到新的broker进行连接的呢。以新的Producer API为例，Producer封装了集群的扩展。



Producer在接收到Broker的返回后，会找到新的Cluster的信息，并更新到发送线程中。

# 进一步需要考虑的问题

## 消息的有序性

同一个Topic 内同一个Partition能保证有序性，因此有需要保证有序的场景，注意使用带有一个partition的Topic。

## 消息的过滤

### Offset与消息回溯

# 比较RocketMQ优缺点

## 可靠性

* RocketMQ支持异步实时刷盘，同步刷盘，同步Replication，异步Replication
* Kafka使用异步刷盘方式，异步Replication

## 性能对比

* [Kafka单机写入TPS约在百万条/秒，消息大小10个字节](http://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines)
* RocketMQ单机写入TPS单实例约7万条/秒，单机部署3个Broker，可以跑到最高12万条/秒，消息大小10个字节

## 消息投递实时性

* Kafka使用短轮询方式，实时性取决于轮询间隔时间
* RocketMQ使用长轮询，同Push方式实时性一致，消息的投递延时通常在几个毫秒。

## 消费失败重试

* Kafka消费失败不支持重试
* RocketMQ消费失败支持定时重试，每次重试间隔时间顺延

## 严格的消息顺序

* Kafka支持消息顺序，但是一台Broker宕机后，就会产生消息乱序。不同的Partition，不能保证消息的顺序。
* RocketMQ支持严格的消息顺序，在顺序消息场景下，一台Broker宕机后，发送消息会失败，但是不会乱序

## 定时消息

* Kafka不支持定时消息
* RocketMQ支持两类定时消息
  + 开源版本RocketMQ仅支持定时Level
  + 阿里云ONS支持定时Level，以及指定的毫秒级别的延时时间

## 分布式事务消息

* Kafka不支持分布式事务消息
* 阿里云ONS支持分布式定时消息，未来开源版本的RocketMQ也有计划支持分布式事务消息

## 消息查询

* Kafka不支持消息查询
* RocketMQ支持根据Message Id查询消息，也支持根据消息内容查询消息（发送消息时指定一个Message Key，任意字符串，例如指定为订单Id）

## 消息回溯

* Kafka理论上可以按照Offset来回溯消息
* RocketMQ支持按照时间来回溯消息，精度毫秒，例如从一天之前的某时某分某秒开始重新消费消息

## 消息轨迹

* Kafka不支持消息轨迹
* 阿里云ONS支持消息轨迹

## Broker端消息过滤

* Kafka不支持Broker端的消息过滤
* RocketMQ支持两种Broker端消息过滤方式
  + 根据Message Tag来过滤，相当于子topic概念
  + 向服务器上传一段Java代码，可以对消息做任意形式的过滤，甚至可以做Message Body的过滤拆分。

## 成熟度

* Kafka在日志领域比较成熟
* RocketMQ在阿里集团内部有大量的应用在使用，每天都产生海量的消息，并且顺利支持了多次天猫双十一海量消息考验，是数据削峰填谷的利器。