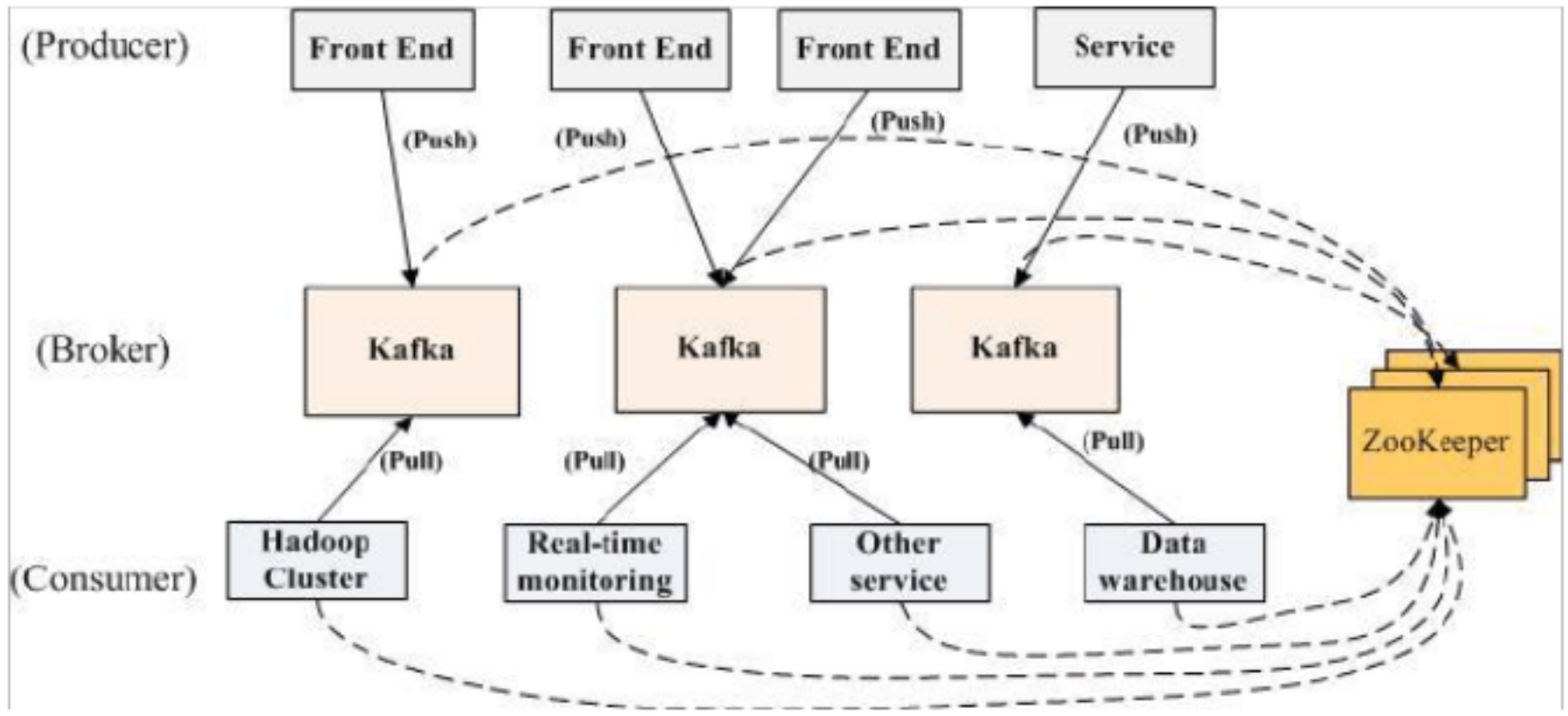
Kafka深度分析

架构

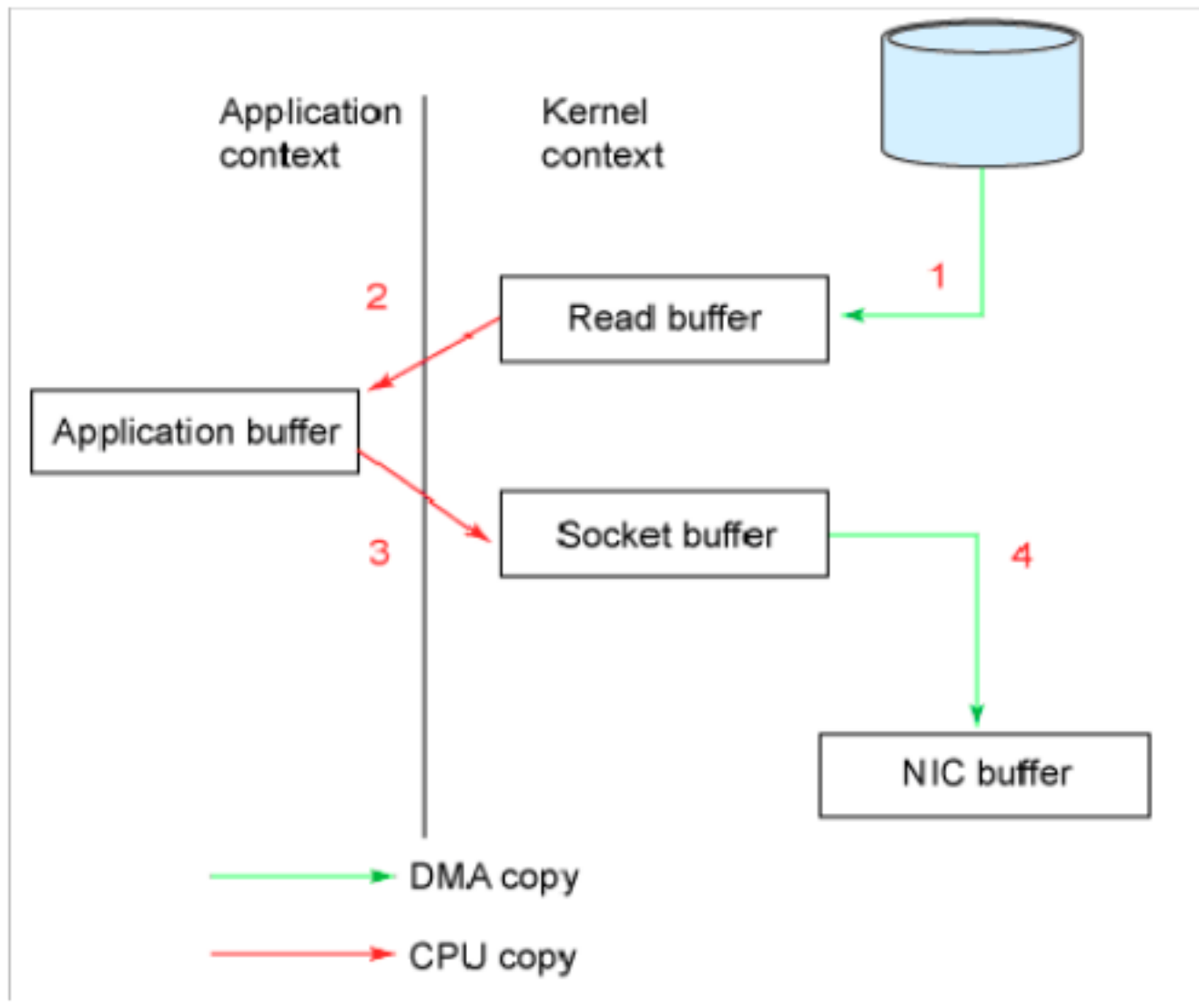


kafka 是显式分布式架构， producer 、broker（Kafka）和 consumer 都可以有多个。 Kafka的运行依赖于 ZooKeeper， Producer 推送消息给 kafka， Consumer 从 kafka 拉消息。

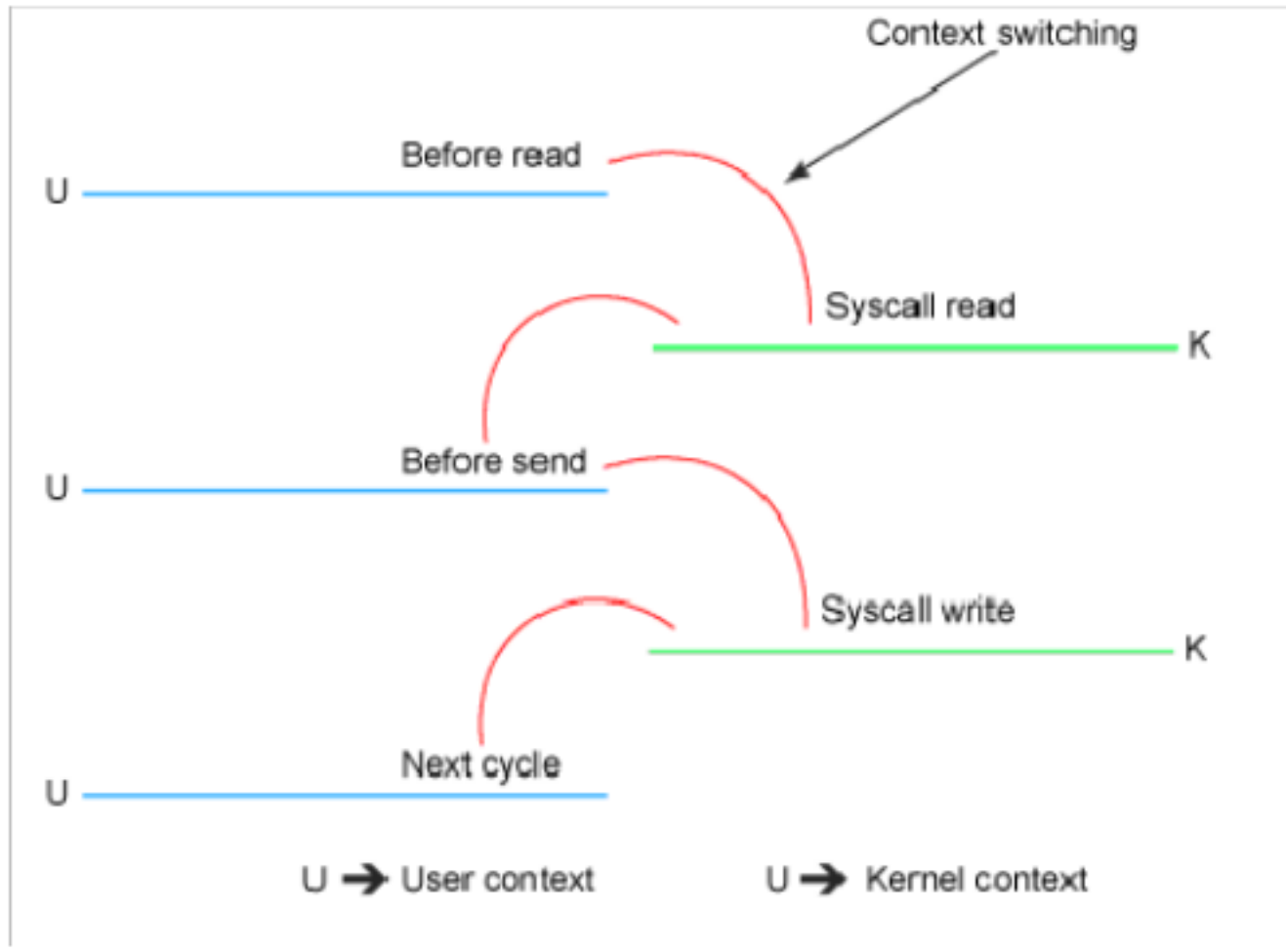
kafka 关键技术点

（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：



Sendfile 系统调用：



（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 Groupid 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 Groupid 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 Groupid 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 ：

```
kafka-console-consumer.sh                kafka-run-class.sh
kafka-console-producer.sh                kafka-server-start.sh
kafka-consumer-offset-checker.sh         kafka-server-stop.sh
kafka-consumer-perf-test.sh              kafka-simple-consumer-shell.sh
kafka-mirror-maker.sh                    kafka-topics.sh
kafka-preferred-replica-election.sh      windows
kafka-producer-perf-test.sh              zookeeper-server-start.sh
kafka-reassign-partitions.sh             zookeeper-server-stop.sh
kafka-replay-log-producer.sh             zookeeper-shell.sh
kafka-replica-verification.sh
```

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.prop
erties &
```

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.pr
operties
>cp ../config/server.properties ../config/server-2.pr
operties
```

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。

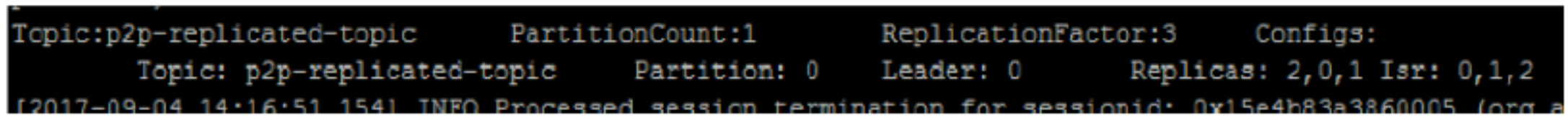
```
>./kafka-server-start.sh ../config/server-1.propertie
s &

>./kafka-server-start.sh ../config/server-2.propertie
s &
```

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
```



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
```

```
Topic:replicated3-partitions3-topic     PartitionCount:3        ReplicationFactor:3     Configs:
        Topic: replicated3-partitions3-topic    Partition: 0    Leader: 1       Replicas: 1,2,0 Isr: 1,2,0
        Topic: replicated3-partitions3-topic    Partition: 1    Leader: 2       Replicas: 2,0,1 Isr: 1,2,0
        Topic: replicated3-partitions3-topic    Partition: 2    Leader: 0       Replicas: 0,1,2 Isr: 1,2,0
```

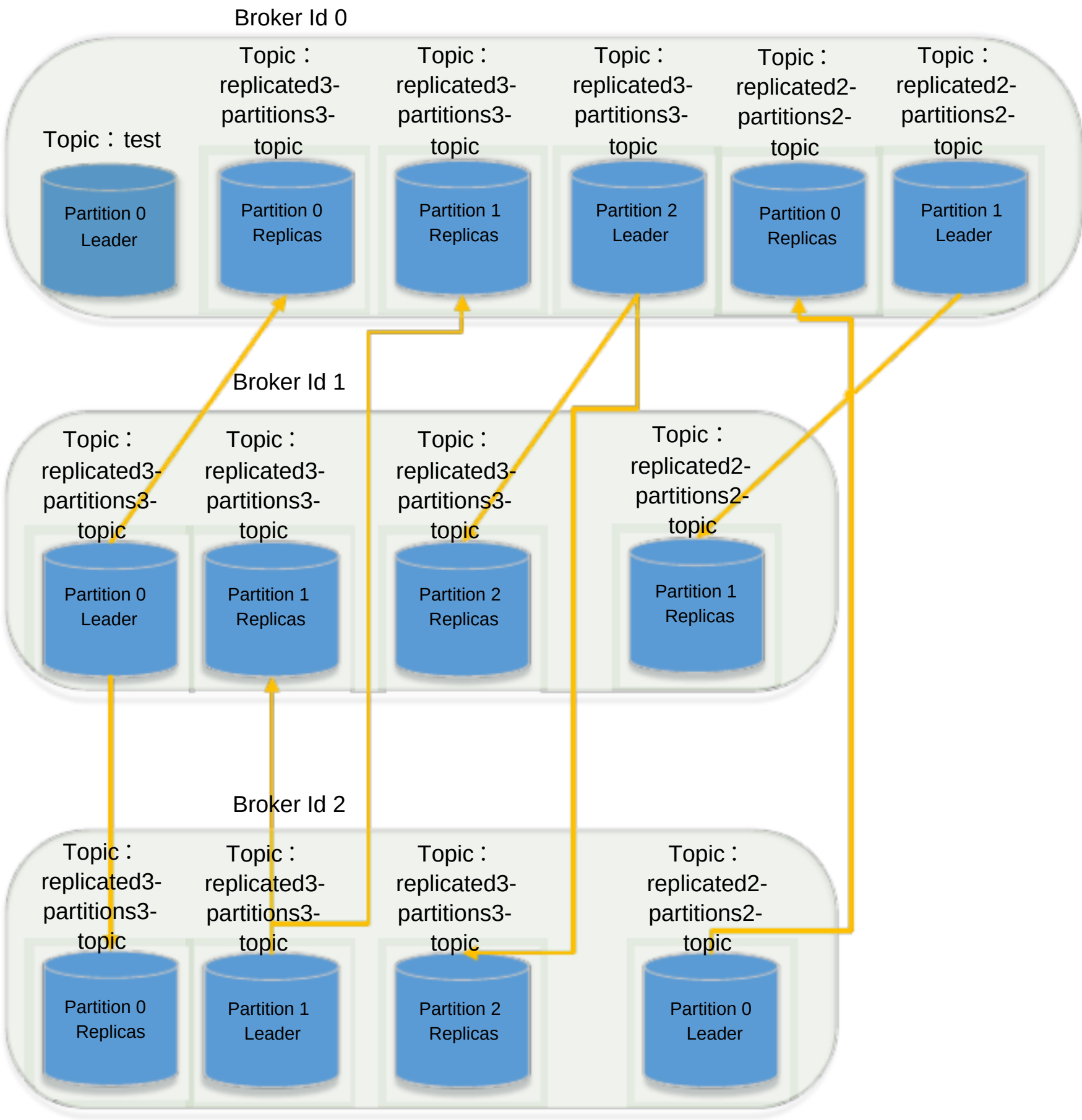
```
>./kafka-topics.sh --describe --zookeeper
localhost:2181 --topic replicated2-partitions3-topic
```

```
Topic:replicated2-partitions2-topic     PartitionCount:2        ReplicationFactor:2     Configs:
        Topic: replicated2-partitions2-topic    Partition: 0    Leader: 2       Replicas: 2,0   Isr: 2,0
        Topic: replicated2-partitions2-topic    Partition: 1    Leader: 0       Replicas: 0,1   Isr: 1,0
```

```
>./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
```

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

```
public static void main(String[] args) {

    // 设置配置属性
    Map<String,String> props = new HashMap<String,String>();
    props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,"10.41.1.121:9092,10.41.1.121:9093,10.41.1.121:9094");
    props.put(ProducerConfig.ACKS_CONFIG, "0");
    props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, "com.deiyida.kafka.serialization.PersonSerializer");
    props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, "org.apache.kafka.common.serialization.StringSerializer");
    long start =System.currentTimeMillis();
    // 创建producer
    Producer<String, Person> producer = new KafkaProducer(props);

    //回调
    Callback callback = new Callback() {
        public void onCompletion(RecordMetadata metadata, Exception e) {
            System.out.println(metadata.offset());
            System.out.println(metadata.partition());
            System.out.println(metadata.topic());

            if (e != null)
                e.printStackTrace();
        }
    };

    Person ps =new Person();
    ps.setAge(11);
    ps.setName("卡卡");

    ProducerRecord<String,Person> record = new ProducerRecord<String,Person>("p2p-replicated-topic","这是什么",ps);
    producer.send(record, callback);
    System.out.println("耗时:" + (System.currentTimeMillis() - start));
    // 关闭producer
```

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 mapof  (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）
        *      @paramumStreams the number of messagestreams 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.

ProducerAPIConfig

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。
				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。
batch.size	int	16384	medium	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

				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。
linger.ms	long	0	medium	
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。

ConsumerAPI 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	是用户特定的字符串，用来在每次请

	value	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

```
// 设置配置属性
Map<String,String> props = new HashMap<String,String>();
props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,"10.41.1.121:9092,10.41.1.121:9093,10.41.1.121:9094");
props.put(ProducerConfig.ACKS_CONFIG, "0");
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, "com.daiyida.kafka.serialization.PersonSerializer");
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, "org.apache.kafka.common.serialization.StringSerializer");
long start =System.currentTimeMillis();
// 创建producer
Producer<String, Person> producer = new KafkaProducer(props);

//回调
Callback callback = new Callback() {
    public void onCompletion(RecordMetadata metadata, Exception e) {
        System.out.println(metadata.offset());
        System.out.println(metadata.partition());
        System.out.println(metadata.topic());

        if (e != null)
            e.printStackTrace();
    }
};

Person ps =new Person();
ps.setAge(11);
ps.setName("卡卡");

ProducerRecord<String,Person> record = new ProducerRecord<String,Person>("p2p-replicated-topic","这是什么",ps);
producer.send(record, callback);
```

KafkaProducerTest.java

具体参看附件

Consumer

创建配置

```
private static ConsumerConfig createConsumerConfig(String a_zookeeper, String a_groupId) {
    Properties props = new Properties();
    props.put("zookeeper.connect", a_zookeeper);
    props.put("group.id", a_groupId);
    props.put("zookeeper.session.timeout.ms", "400");
    props.put("zookeeper.sync.time.ms", "200");
    props.put("auto.commit.interval.ms", "1000");

    return new ConsumerConfig(props);
```

创建 Consumer 对象

```
consumer = kafka.consumer.Consumer.createJavaConsumerConnector(
        createConsumerConfig(a_zookeeper, a_groupId));
```

创建 KafakaStream 对象

```
Map<String, Integer> topicCountMap = new HashMap<String, Integer>();
topicCountMap.put(topic, new Integer(a_numThreads));
Map<String, List<KafkaStream<String, Person>>> consumerMap = consumer
        .createMessageStreams(topicCountMap, new StringDecoder(null),
                new Decoder<Person>() {

                    public Person fromBytes(byte[] arg0) {
                        ObjectInputStream ois;
                        Person ps = null;
                        try {
                            ois = new ObjectInputStream(
                                    new ByteArrayInputStream(arg0));
                            ps = (Person) ois.readObject();
                        } catch (IOException e) {
                            throw new RuntimeException(e);
                        } catch (ClassNotFoundException e) {
                            throw new RuntimeException(e);
                        }
                        return ps;
                    }
                });
List<KafkaStream<String, Person>> streams = consumerMap.get(topic);
```

读取消息

```
ConsumerIterator<String, Person> it = m_stream.iterator();
while (it.hasNext()) {
    MessageAndMetadata<String, Person> next = it.next();

    Person message = next.message();
    System.out
            .printf("Thread %d -offset is %d ; -key is %s; -Person Name is %s \n",
                    m_threadNumber, next.offset(), next.key(),
                    message.getName());
}
System.out.println("Shutting down Thread: " + m_threadNumber);
```


KafkaConsumerTest.java


HighLevelConsumer.java

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 ivycache，通过这种方式实现 kafkamanager 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

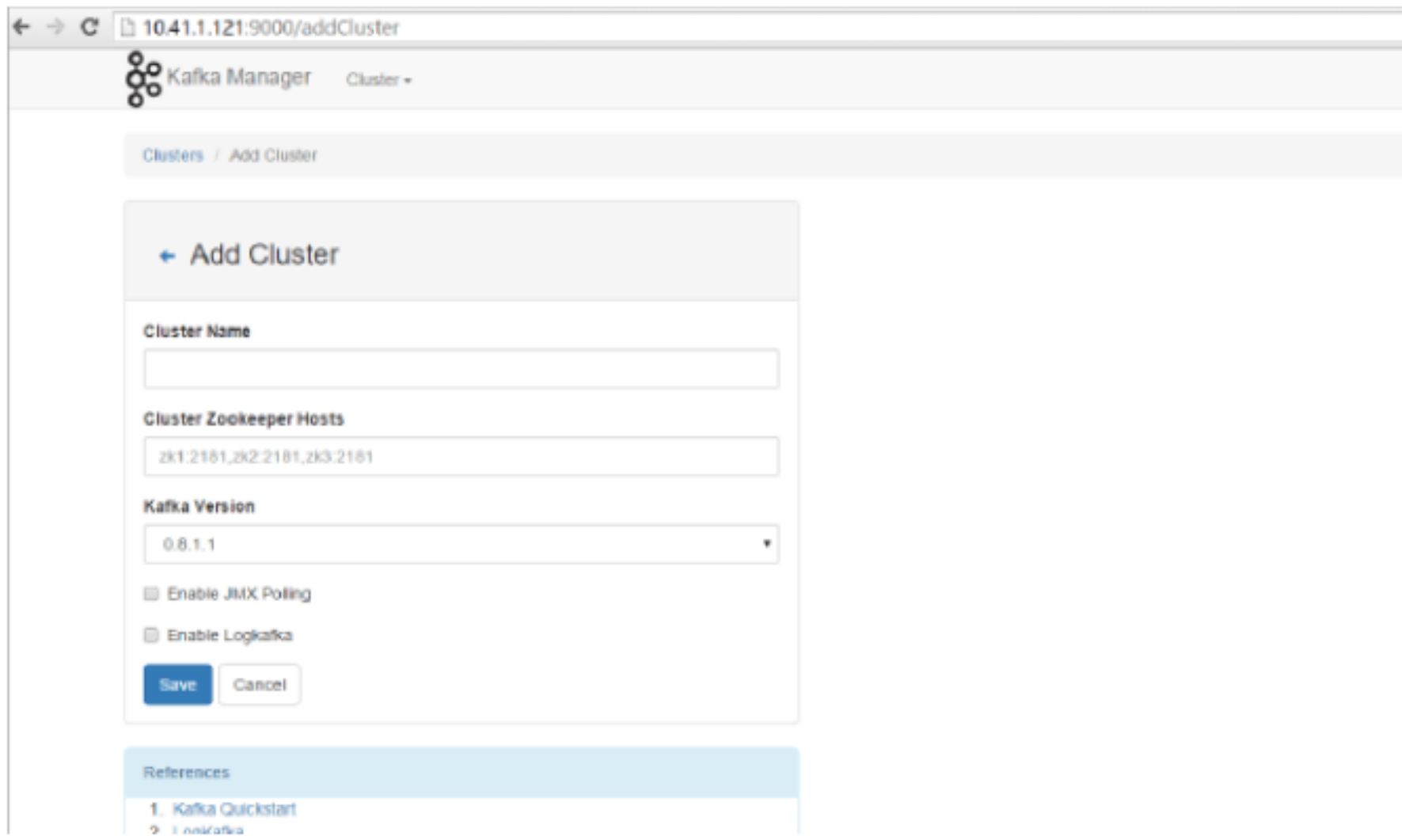
进入 bin 目录下执行命令：

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

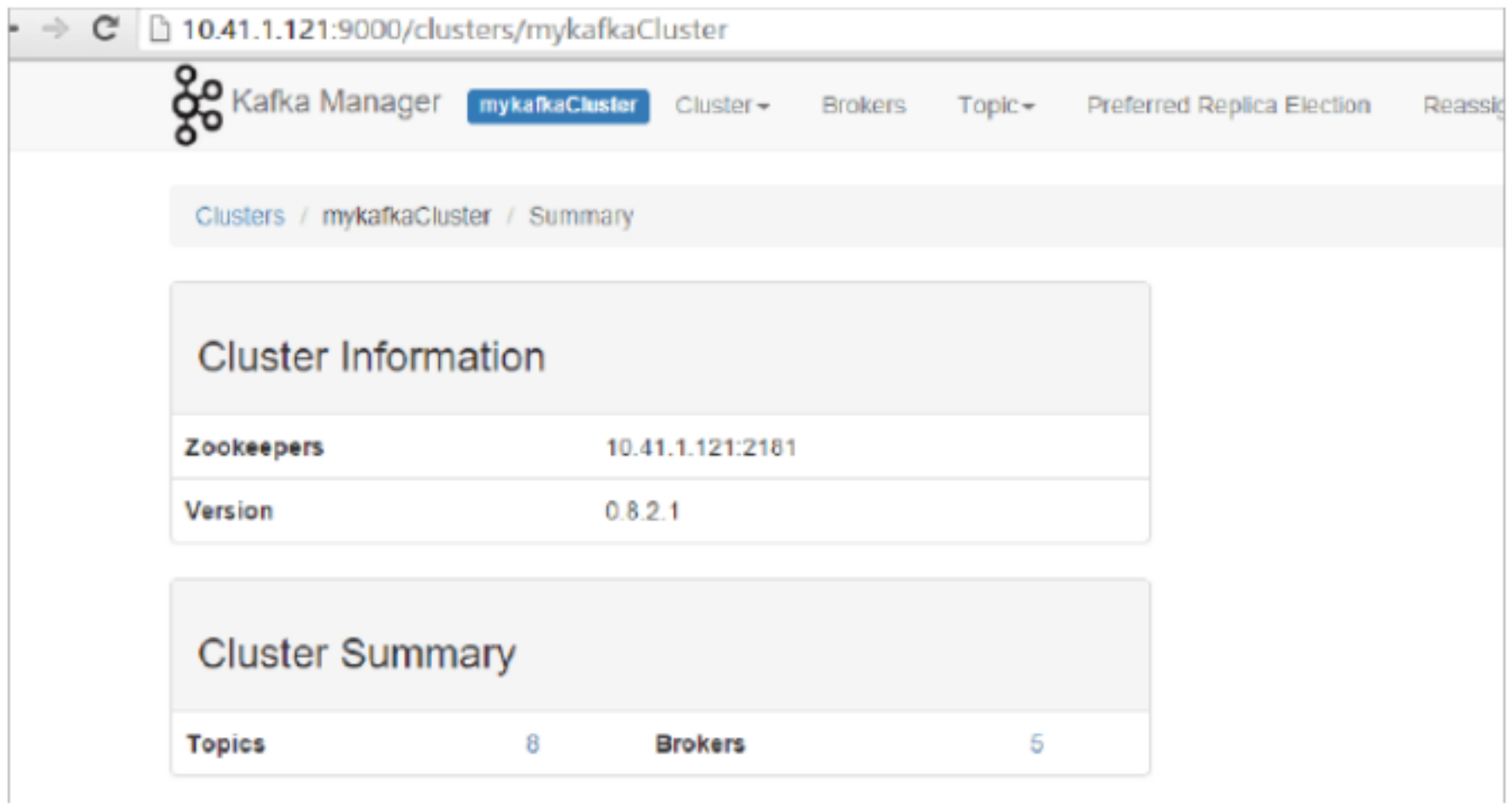
如果命令不能执行，请先给文件授权：

```
.chmod 777 kafka-manager
```

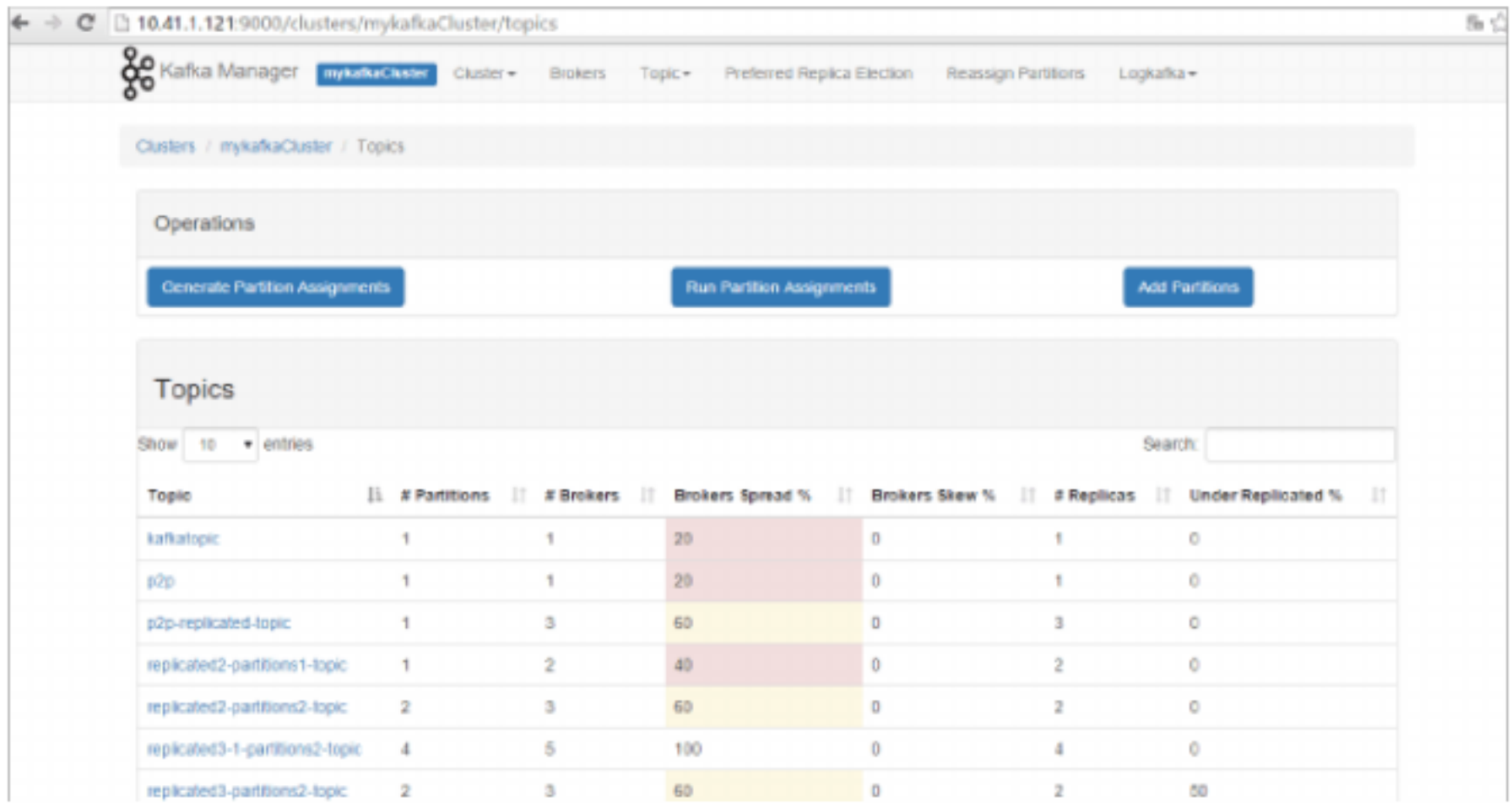
打开浏览器输入 http://x.x.x.x:9000 ，即可进入 kafkamanager 的管理页面。



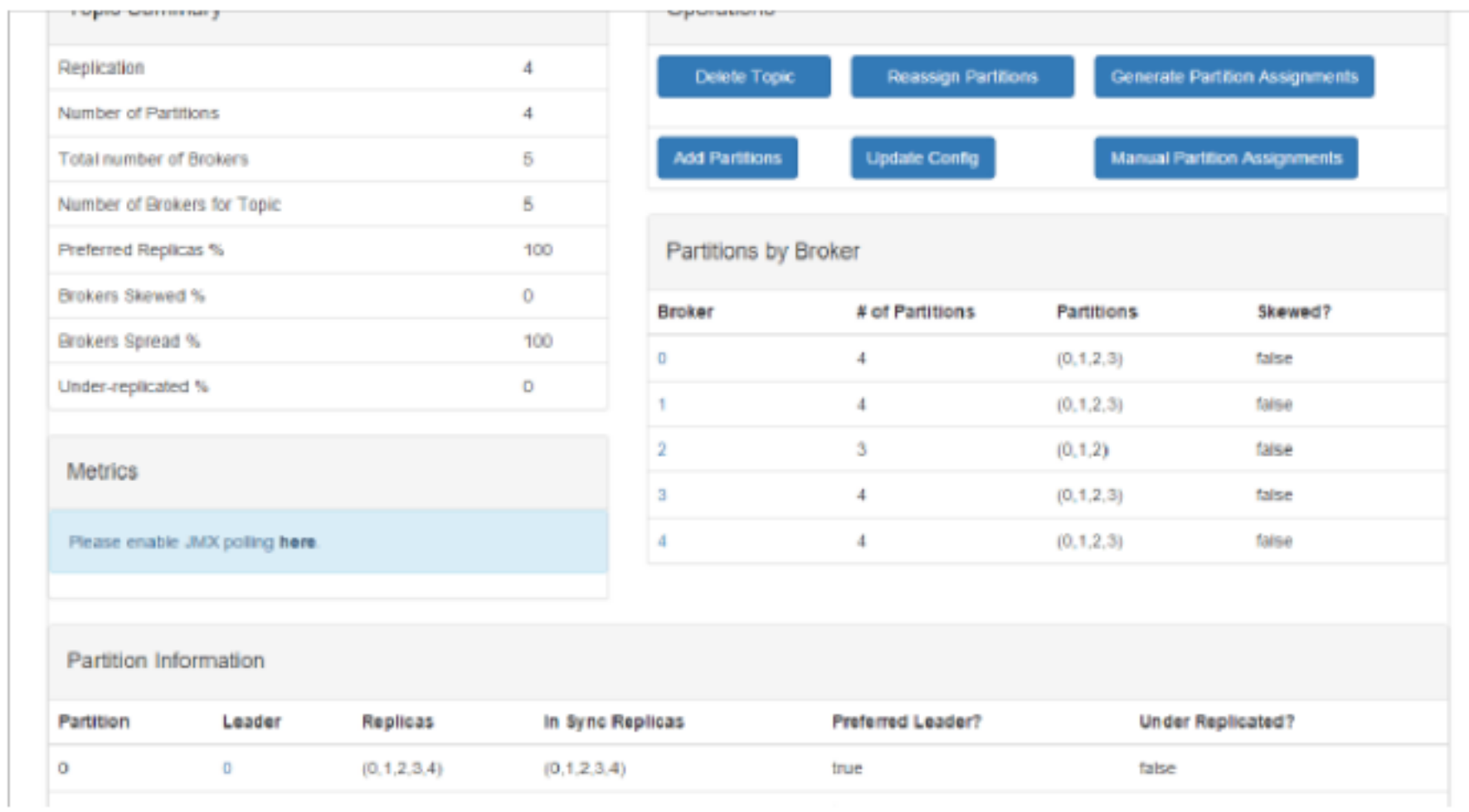
填写集群名字， zk 的地址和端口， 选择 kafka 的版本，我测试用的版本是 0.8.2.1。然后保存。



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：

a) --generate: 对指定的 topics 移动所有的 partition 到新的 brokers
b) --execute: 指定 --reassignment-json-file 用自定义的重新分配方案
c) --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

```
Topic:replicated3-partitions3-topic     PartitionCount:4        ReplicationFactor:4     Configs:
        Topic: replicated3-partitions3-topic    Partition: 0    Leader: 1       Replicas: 1,2,0         Isr: 0,1,2
        Topic: replicated3-partitions3-topic    Partition: 1    Leader: 2       Replicas: 2,0,1         Isr: 0,1,2
        Topic: replicated3-partitions3-topic    Partition: 2    Leader: 0       Replicas: 0,1,2         Isr: 0,1,2
```

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 –partitions4

执行命令后，查看状态如下

```
Topic:replicated3-partitions3-topic     PartitionCount:4        ReplicationFactor:4     Configs:
        Topic: replicated3-partitions3-topic    Partition: 0    Leader: 1       Replicas: 1,2,0         Isr: 0,1,2
        Topic: replicated3-partitions3-topic    Partition: 1    Leader: 2       Replicas: 2,0,1         Isr: 0,1,2
        Topic: replicated3-partitions3-topic    Partition: 2    Leader: 0       Replicas: 0,1,2         Isr: 0,1,2
        Topic: replicated3-partitions3-topic    Partition: 3    Leader: 0       Replicas: 0,2,3         Isr: 0,2,1
```

2、 使用重新分配工具，创建重新分配 json 文件

> Vimtopics-to-move.json

```
{"version":1,
"partitions":[{"topic":"replicated3-partitions3-topic","partition":0,"replicas":[1,2,0,3]},
              {"topic":"replicated3-partitions3-topic","partition":1,"replicas":[2,0,1,3]},
              {"topic":"replicated3-partitions3-topic","partition":2,"replicas":[0,1,2,3]},
              {"topic":"replicated3-partitions3-topic","partition":3,"replicas":[0,2,3,1]}
             ]
}
~
```

3、 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

```
Status of partition reassignment:
Reassignment of partition [replicated3-partitions3-topic,0] is still in progress
Reassignment of partition [replicated3-partitions3-topic,1] is still in progress
Reassignment of partition [replicated3-partitions3-topic,2] is still in progress
Reassignment of partition [replicated3-partitions3-topic,3] completed successfully
```

4、 再次查看 replicated3-partitions3-topic 的状态

```
Topic:replicated3-partitions3-topic     PartitionCount:4        ReplicationFactor:4     Configs:
        Topic: replicated3-partitions3-topic    Partition: 0    Leader: 1       Replicas: 1,2,0,3       Isr: 0,1,2
        Topic: replicated3-partitions3-topic    Partition: 1    Leader: 2       Replicas: 2,0,1,3       Isr: 0,1,2
        Topic: replicated3-partitions3-topic    Partition: 2    Leader: 0       Replicas: 0,1,2,3       Isr: 0,1,2
        Topic: replicated3-partitions3-topic    Partition: 3    Leader: 0       Replicas: 0,2,3,1       Isr: 0,2,1
```

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。

```
private void handleMetadataResponse(RequestHeader header, Struct body, long now) {
    this.metadataFetchInProgress = false;
    MetadataResponse response = new MetadataResponse(body);
    Cluster cluster = response.cluster();
    // don't update the cluster if there are no valid nodes...the topic we want may
    // created which means we will get errors and no nodes until it exists
    if (cluster.nodes().size() > 0) {
        this.metadata.update(cluster, now);
    } else {
        log.trace("Ignoring empty metadata response with correlation id {}.", heade
        this.metadata.failedUpdate(now);
    }
}
```

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
- 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。