Kafka

kafka是什么？kafka仅仅是属于消息 中间件吗?

kafka在设计之初的时候 开发人员们在除了消息中间件以外，还想吧kafka设计为一个能够存储数据的系统，有点像常见的非关系型数据库，比如说NoSql等。除此之外 还希望kafka能支持持续变化，不断增长的数据流, 可以发布和订阅数据流，还可以对于这些数据进行保存

也就是说kafka的本质 是一个数据存储平台，流平台， 只是他在做消息发布，消息消费的时候我们可以把他当做消息中间件来用。

而且kafka在设计之初就是采用分布式架构设计的， 基于集群的方式工作，且可以自由伸缩，所以 kafka构建集群非常简单

基本概念:

- Broker : 和AMQP里协议的概念一样， 就是消息中间件所在的服务器
- Topic(主题) : 每条发布到Kafka集群的消息都有一个类别，这个类别被称为Topic。（物理上不同Topic的消息分开存储，逻辑上一个Topic的消息虽然保存于一个或多个broker上但用户只需指定消息的Topic即可生产或消费数据而不必关心数据存于何处）
- Partition(分区) : Partition是物理上的概念，体现在磁盘上面，每个Topic包含一个或多个Partition.
- Producer : 负责发布消息到Kafka broker
- Consumer : 消息消费者，向Kafka broker读取消息的客户端。
- Consumer Group（消费者群组） : 每个Consumer属于一个特定的Consumer Group（可为每个Consumer指定group name，若不指定group name则属于默认的group）。
- offset 偏移量： 是kafka用来确定消息是否被消费过的标识，在kafka内部体现就是一个递增的数字

kafka消息发送的时候 ,考虑到性能 可以采用打包方式发送， 也就是说 传统的消息是一条一条发送， 现在可以先把需要发送的消息缓存在客户端， 等到达一定数值时， 再一起打包发送， 而且还可以对发送的数据进行压缩处理，减少在数据传输时的开销

kafka优缺点

优点: 基于磁盘的数据存储 高伸缩性 高性能 应用场景 : 收集指标和日志 提交日志 流处理

缺点:

运维难度大 偶尔有数据混乱的情况 对zookeeper强依赖 多副本模式下对带宽有一定要求

kafka安装&启动

kafka安装的话，直接 从官网下载压缩包下来解压就可以了

注意的是， 启动kafka要先启动zookeeper kafka默认自带了zookeeper 可以启动他自带的 也可以自己另外使用

启动kafka需要执行 kafka-server-start.bat 文件 然后 需要传入一个路径参数 就是你server.config文件的地址 一般情况下传入../../config/server.properties 即可

刚刚提到的zookeeper kafka默认的zookeeper 启动的话启动zookeeper-server-start.bat文件即可 同样 要传入路径参数：../../config/zookeeper.properties

server 参数解释：

log.dirs: 日志文件存储地址， 可以设置多个

num.recovery.threads.per.data.dir：用来读取日志文件的线程数量，对应每一个log.dirs 若此参数为2 log.dirs 为2个目录 那么就会有4个线程来读取

auto.create.topics.enable:是否自动创建tiopic

num.partitions: 创建topic的时候自动创建多少个分区 (可以在创建topic的时候手动指定)

log.retention.hours: 日志文件保留时间 超时即删除

log.retention.bytes: 日志文件最大大小

log.segment.bytes: 当日志文件达到一定大小时，开辟新的文件来存储(分片存储)

log.segment.ms: 同上 只是当达到一定时间时 开辟新的文件

message.max.bytes: broker能接收的最大消息大小(单条) 默认1M

kafka基本管理操作命令:

##列出所有主题 kafka-topics.bat --zookeeper localhost:2181/kafka --list

##列出所有主题的详细信息 kafka-topics.bat --zookeeper localhost:2181/kafka --describe

##创建主题 主题名 my-topic，1副本，8分区 kafka-topics.bat --zookeeper localhost:2181/kafka --create --replication-factor 1 --partitions 8 --topic my-topic

##增加分区，注意：分区无法被删除 kafka-topics.bat --zookeeper localhost:2181/kafka --alter --topic my-topic --partitions 16

##删除主题 kafka-topics.bat --zookeeper localhost:2181/kafka --delete --topic my-topic

##列出消费者群组（仅Linux） kafka-topics.sh --new-consumer --bootstrap-server localhost:9092/kafka --list

##列出消费者群组详细信息（仅Linux） kafka-topics.sh --new-consumer --bootstrap-server localhost:9092/kafka --describe --group 群组名

kafka java客户端实战

引入maven依赖:

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>0.11.0.0</version>
</dependency>
```

注意 我这里已经创建了一个叫 test-topic 的主题 如果你们没创建先创建后再执行代码

生产者:

```
public class TestProducter {

    public static void main(String[] args) throws  Exception{
        Properties properties = new Properties();
        //指定kafka服务器地址 如果是集群可以指定多个  但是就算只指定一个他也会去集群环境下寻找其他的节点地 址
        properties.setProperty("bootstrap.servers","127.0.0.1:9092");
        //key序列化器
        properties.setProperty("key.serializer", StringSerializer.class.getName());
        //value序列化器
        properties.setProperty("value.serializer",StringSerializer.class.getName());
        KafkaProducer<String,String> kafkaProducer = new KafkaProducer<String, String>(properties);
        ProducerRecord<String, String> stringStringProducerRecord = new ProducerRecord<String, String>("test-topic",1,"testKey","hello");
        Future<RecordMetadata> send = kafkaProducer.send(stringStringProducerRecord);
        RecordMetadata recordMetadata = send.get();
        System.out.println(recordMetadata);
    }

}
```

消费者:

```
public class TestCousmer {

    public static void main(String[] args) {
        Properties properties = new Properties();
        properties.setProperty("bootstrap.servers","127.0.0.1:9092");
        properties.setProperty("key.deserializer", StringDeserializer.class.getName());

 properties.setProperty("value.deserializer",StringDeserializer.class.getName());
        properties.setProperty("group.id","1111");
        KafkaConsumer<String,String> consumer = new KafkaConsumer<String, String>
(properties);
        consumer.subscribe(Collections.singletonList("test-topic"));

        while (true){
            ConsumerRecords<String, String> poll = consumer.poll(500);
            for (ConsumerRecord<String, String> stringStringConsumerRecord : poll) {
                System.out.println(stringStringConsumerRecord);
            }
        }
    }
}
```

kafka生产者参数详解

acks:

至少要多少个分区副本接收到了消息返回确认消息 一般是 0:只要消息发送出去了就确认(不管是否失败) 1:只要 有一个broker接收到了消息 就返回 all： 所有集群副本都接收到了消息确认 当然 2 3 4 5 这种数字都可以， 就是具体多少台机器接收到了消息返回， 但是一般这种情况很少用到

buffer.memory：

生产者缓存在本地的消息大小： 如果生产者在生产消息的速度过快 快过了往 broker发送消息的速度 那么就会出现buffer.memory不足的问题 默认值为32M 注意 单位是byte 大概3355000左右

max.block.ms:

生产者获取kafka元数据(集群数据，服务器数据等) 等待时间： 当因网络原因导致客户端与服务器通讯时等待的时间超过此值时 会抛出一个TimeOutExctption 默认值为 60000ms

retries：

设置生产者生产消息失败后重试的次数 默认值 3次

retry.backoff.ms:

设置生产者每次重试的间隔 默认 100ms

batch.size:

生产者批次发送消息的大小 默认16k 注意单位还是byte

linger.ms:

生产者生产消息后等待多少毫秒发送到broker 与batch.size 谁先到达就根据谁 默认值为0

compression.type:

kafka在压缩数据时使用的压缩算法 可选参数有:none、gzip、snappy none即不压缩 gzip,和snappy压缩算法之间取舍的话 gzip压缩率比较高 系统cpu占用比较大 但是带来的好处是 网络带宽占用少， snappy压缩比没有gzip高 cpu占用率不是很高 性能也还行， 如果网络带宽比较紧张的话 可以选择gzip 一般推荐snappy

client.id:

一个标识， 可以用来标识消息来自哪， 不影响kafka消息生产

max.in.flight.requests.per.connection:

指定kafka一次发送请求在得到服务器回应之前,可发送的消息数量

偏移量与偏移量提交

偏移量是kafka特别重要的一个概念特别是在消费者端， 我们之前也有简单提到过偏移量是拿来干嘛的.

偏移量是一个自增长的ID 用来标识当前分区的哪些消息被消费过了， 这个ID会保存在kafka的broker当中 而且 消费者本地也会存储一份 因为每次消费每一条消息都要更新一下偏移量的话 难免会影响整个broker的吞吐量 所以一般 这个偏移量在每次发生改动时 先由消费者本地改动， 默认情况下 消费者每五秒钟会提交一次改动的偏移量，这样做虽然说吞吐量上来了， 但是可能会出现重复消费的问题: 因为可能在下一次提交偏移量之前 消费者本地消费了一些消息，然后发生了分区再均衡(分区再均衡在下面有讲) 那么就会出现一个问题 假设上次提交的偏移量是2000 在下一次提交之前 其实消费者又消费了500条数据 也就是说当前的偏移量应该是2500 但是这个2500只在消费者本地， 也就是说 假设其他消费者去消费这个分区的时候拿到的偏移量是2000 那么又会从2000开始消费消息 那么 2000到2500之间的消息又会被消费一遍,这就是重复消费的问题.

kafka对于这种问题也提供了解决方案:手动提交

你可以关闭默认的自动提交(enable.auto.commit= false) 然后使用kafka提供的API来进行偏移量提交: 卡夫卡提供了两种方式提交你的偏移量 :同步和异步

```
//同步提交偏移量
kafkaConsumer.commitSync();
//异步提交偏移量
kafkaConsumer.commitAsync();
```

他们之间的区别在于 同步提交偏移量会等待服务器应答 并且遇到错误会尝试重试，但是会一定程度上影响性能不过能确保偏移量到底提交成功与否

而异步提交的对于性能肯定是有提示的 但是弊端也就像我们刚刚所提到 遇到错误没办法重试 因为可能在收到你这个结果的时候又提交过偏移量了 如果这时候重试 又会导致消息重复的问题了..

其实 我们可以采用同步+异步的方式来保证提交的正确性以及服务器的性能

因为 异步提交的话 如果出现问题但是不是致命问题的话 可能下一次提交就不会出现这个问题了， 所以 有些异常是不需要解决的(可能单纯的就是网络抽风了呢?) 所以 我们平时可以采用异步提交的方式 等到消费者中断了(遇到了致命问题，或是强制中断消费者) 的时候再使用同步提交(因为这次如果失败了就没有下次了... 所以要让他重试)。

具体代码:

```
try {
    while (true) {
        ConsumerRecords<String, String> poll = kafkaConsumer.poll(500);
        for (ConsumerRecord<String, String> context : poll) {
            System.out.println("消息所在分区:" + context.partition() + "-消息的偏移量:" +
context.offset()
                    + "key:" + context.key() + "value:" + context.value());
        }
        //正常情况异步提交
         kafkaConsumer.commitAsync();
    }
} catch (Exception e) {
    e.printStackTrace();
} finally {
    try {
        //当程序中断时同步提交
        kafkaConsumer.commitSync();
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        //关闭当前消费者  具体在下面有讲
        kafkaConsumer.close();
    }

}
```

值得一提的是 在手动提交时kafka提供了你可以传入具体的偏移量来完成提交 也就是指定偏移量提交,但是非常不建议手动指定 因为如果指定的偏移量 小于 分区所存储的偏移量大小的话 那么会导致消息重复消费， 如果指定的偏移量大于分区所存储的偏移量的话，那么会导致消息丢失.

代码:

```
Map<TopicPartition, OffsetAndMetadata> offset = new HashMap<>();
//我这里就指定了test-topic这个主题下的分区1  OffsetAndMetadata:第一个参数为你要提交的偏移量 第二个参数可以选择性的传入业务ID 可以拿来确定这次提交  这里我直接提交偏移量为0 那么会导致下个消费者或者说分区再均衡之后再来读取这个分区的数据会从第一条开始读取
offset.put(new TopicPartition("test-topic", 1), new OffsetAndMetadata(0, "1"));
//指定偏移量提交 参数为map集合  key为指定的主题下的分区, value 为你要提交的偏移量
kafkaConsumer.commitSync(offset);
```

Rebalance 分区再均衡

这也是kafka里面非常重要的一个概念

首先 Rebalance 是一个操作 在以下情况下会触发Rebalance 操作:

1. 组成员发生变更(新consumer加入组、已有consumer主动离开组或已有consumer崩溃了)
2. 订阅主题数发生变更，如果你使用了正则表达式的方式进行订阅，那么新建匹配正则表达式的topic就会触发rebalance
3. 订阅主题的分区数发生变更

当触发Rebalance kafka重新分配分区所有权

何为分区所有权？ 我们之前有提到过， 消费者有一个消费者组的概念， 而且一个消费者组在消费一个主题时有以下规则 一个消费者可以消费多个分区 但是一个分区只能被一个消费者消费 如果 我有分区 0 1 2 现在有消费者 A，B 那么 kafka可能会让消费者A 消费 0，1 这2个分区 那么 这时候 我们就会说 消费者A 拥有 分区 0,1的所有权。

当触发 Rebalance 的时候 kafka会重新分配这个所有权 还是基于刚刚的比方 消费者A 拥有 0 和1 的所有权 消费者B 会有2的所有权 当消费者B离开kafka的时候 这时候 kafka会重新分配一下所有权 此时 整个消费者组只有一个A 那么 0 1 2 三个分区的所有权都会属于A 同理 如果这时候有消费者C进入这个消费者组 那么 这时候kafka会确保每一个消费者都能消费一个分区.

当触发Rebalance时 由于kafka正在分配所有权 会导致消费者不能消费， 而且 还会引发一个重复消费的问题， 当

消费者还没来得及提交偏移量时 分区所有权遭到了重新分配 那么这时候就会导致一个消息被多个消费者重复消费

那么 解决方案就是 在消费者订阅时， 添加一个再均衡监听器， 也就是当kafka在做Rebalance 操作前后 均会调用再均衡监听器 那么这时候 我们可以在kafka Rebalance之前提交我们消费者最后处理的消息来解决这个问题。

Close（）：

当我们不需要某个消费者继续消费kafka当中的数据时， 我们可以选择调用Close方法来关闭它，在关闭之前 close方法会发送一个通知告诉kafka我这个消费者要退出了， 那么 kafka就会准备Rebalance 而且如果是采用的自动提交偏移量 消费者自身也会在关闭自己之前提交最后所消费的偏移量 。

当然 即使没有调用close方法 而是直接强制中断了消费者的进程 kafka也会根据我们后面会说到的系统参数捕捉到消费者退出了。

独立消费者:

kafka支持这样的需求： 可能你的消费者不想订阅某个主题 也不想加入什么消费组 只想订阅某个(多个)主题下的某个（多个）分区。

那么可以采用分配的方式， 而不是订阅 ， 我们之前讲的都是基于消费组订阅某个主题来完成消息的消费， 那么你订阅的主题有哪些分区的消息是属于你的 这个是kafka来分配的 而不是你自己决定的 那么我们可以换为自己分配的方式来完成消息的消费:

```
List<TopicPartition> list = new ArrayList<>();
//new出一个分区对象 声明这个分区是哪个topic下面的哪个分区
list.add(new TopicPartition("test-topic",0));
//分配这个消费者所需要消费的分区，传入一个分区对象集合
kafkaConsumer.assign(list);
```

消费者参数:

fetch.min.bytes:

该属性指定了消费者从服务器获取记录的最小字节数。broker 在收到消费者的数据请求时，如果可用的数据量小于 fetch.min.bytes 指定的大小，那么它会等到有足够的可用数据时才把它返回给消费者。这样可以降低消费者和 broker 的工作负载，因为它们在主题不是很活跃的时候（或者一天里的低谷时段）就不需要来来回回地处理消息。如果没有很多可用数据，但消费者的 CPU 使用率却很高，那么就需要把该属性的值设得比默认值大。如果消费者的数量比较多，把该属性的值设置得大一点可以降低 broker 的工作负载。 默认值为1 byte

fetch.max.wait.ms

我们通过 fetch.min.bytes 告诉 Kafka，等到有足够的数据时才把它返回给消费者。而 feth.max.wait.ms 则用于指定 broker 的等待时间，默认是如果没有足够的数据流入Kafka，消费者获取最小数据量的要求就得不到满足，最终导致 500ms 的延迟。如果 fetch.max.wait.ms 被设为 100ms，并且 fetch.min.bytes 被设为 1MB，那么 Kafka 在收到消费者的请求后，要么返回 1MB 数据，要么在 100ms 后返回所有可用的数据，就看哪个条件先得到满足。默认值为500ms

max.partition.fetch.bytes

该属性指定了服务器从每个分区里返回给消费者的最大字节数。默认值是 1MB

session.timeout.ms 和heartbeat.interval.ms

session.timeout.ms :

消费者多久没有发送心跳给服务器服务器则认为消费者死亡/退出消费者组 默认值:10000ms

heartbeat.interval.ms :

消费者往kafka服务器发送心跳的间隔 一般设置为session.timeout.ms的三分之一 默认值:3000ms

auto.offset.reset:

当消费者本地没有对应分区的offset时 会根据此参数做不同的处理 默认值为:latest

earliest

当各分区下有已提交的offset时，从提交的offset开始消费；无提交的offset时，从头开始消费

latest

当各分区下有已提交的offset时，从提交的offset开始消费；无提交的offset时，消费新产生的该分区下的数据

none

topic各分区都存在已提交的offset时，从offset后开始消费；只要有一个分区不存在已提交的offset，则抛出异常

enable.auto.commit

该属性指定了消费者是否自动提交偏移量，默认值是 true。为了尽量避免出现重复数据和数据丢失，可以把它设为 false，由自己控制何时提交偏移量。如果把它设为 true，还可以通过配置 auto.commit.interval.ms 属性来控制提交的频率。

partition.assignment.strategy

PartitionAssignor 根据给定的消费者和主题，决定哪些分区应该被分配给哪个消费者。Kafka 有两个默认的分配策略。

- Range：该策略会把主题的若干个连续的分区分配给消费者。假设消费者 C1 和消费者 C2 同时订阅了主题 T1 和主题 T2，并且每个主题有 3 个分区。那么消费者 C1 有可能分配到这两个主题的分区 0 和分区 1，而消费者 C2 分配到这两个主题的分区2。因为每个主题拥有奇数个分区，而分配是在主题内独立完成的，第一个消费者最后分配到比第二个消费者更多的分区。只要使用了 Range 策略，而且分区数量无法被消费者数量整除，就会出现这种情况。
- RoundRobin：该策略把主题的所有分区逐个分配给消费者。如果使用 RoundRobin 策略来给消费者 C1 和消费者 C2 分配分区，那么消费者 C1 将分到主题 T1 的分区 0 和分区 2 以及主题 T2 的分区 1，消费者 C2 将分配到主题 T1 的分区 1 以及主题 T2 的分区 0 和分区 2。一般来说，如果所有消费者都订阅相同的主题（这种情况很常见），RoundRobin 策略会给所有消费者分配相同数量的分区（或最多就差一个分区）。

max.poll.records

单次调用 poll() 方法最多能够返回的记录条数 ,默认值 500

receive.buffer.bytes 和 send.buffer.bytes

receive.buffer.bytes 默认值 64k 单位 bytes

send.buffer.bytes 默认值 128k 单位 bytes

这两个参数分别指定了 TCP socket 接收和发送数据包的缓冲区大小。如果它们被设为 -1

使用java来操作kafka管理命令

首先 得引入一个依赖:

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka_2.10</artifactId>
    <version>0.10.2.1</version>
</dependency>
```

我们之前所引入的依赖是kafka客户端的依赖 这个是另外的依赖 不是一回事

创建topic

```
public static void createTopic(){
    ZkUtils zkUtils = ZkUtils.apply("localhost:2181/kafka", 30000, 30000,
JaasUtils.isZkSecurityEnabled());
    System.out.println(JaasUtils.isZkSecurityEnabled());
    AdminUtils.createTopic(zkUtils, "t1", 1, 1, new Properties(),
AdminUtils.createTopic$default$6());
    zkUtils.close();
}
```

删除topic

```
public static  void deleteTopic(){
    ZkUtils zkUtils = ZkUtils.apply("localhost:2181/kafka", 30000, 30000,
JaasUtils.isZkSecurityEnabled());
    AdminUtils.deleteTopic(zkUtils, "t1");
    zkUtils.close();
}
```

列出所有topic

```
public  static void listTopic(){
    ZkUtils zkUtils = ZkUtils.apply("localhost:2181/kafka", 30000, 30000,
JaasUtils.isZkSecurityEnabled());
    List<String> list = JavaConversions.seqAsJavaList(zkUtils.getAllTopics());
    for (String s : list) {
        System.out.println(s);
    }
    zkUtils.close();
}
```