


Kafka可视化管理工具kafka-manager

Kafka Manager

kafka-cluster-1

Cluster

Brokers

Topic

Preferred Replica Election

Reassign Partitions

Consumers

Clusters / kafka-cluster-1 / Topics

Operations

Generate Partition AssignmentsRun Partition AssignmentsAdd Partitions

Topics

Show10entriesSearch:

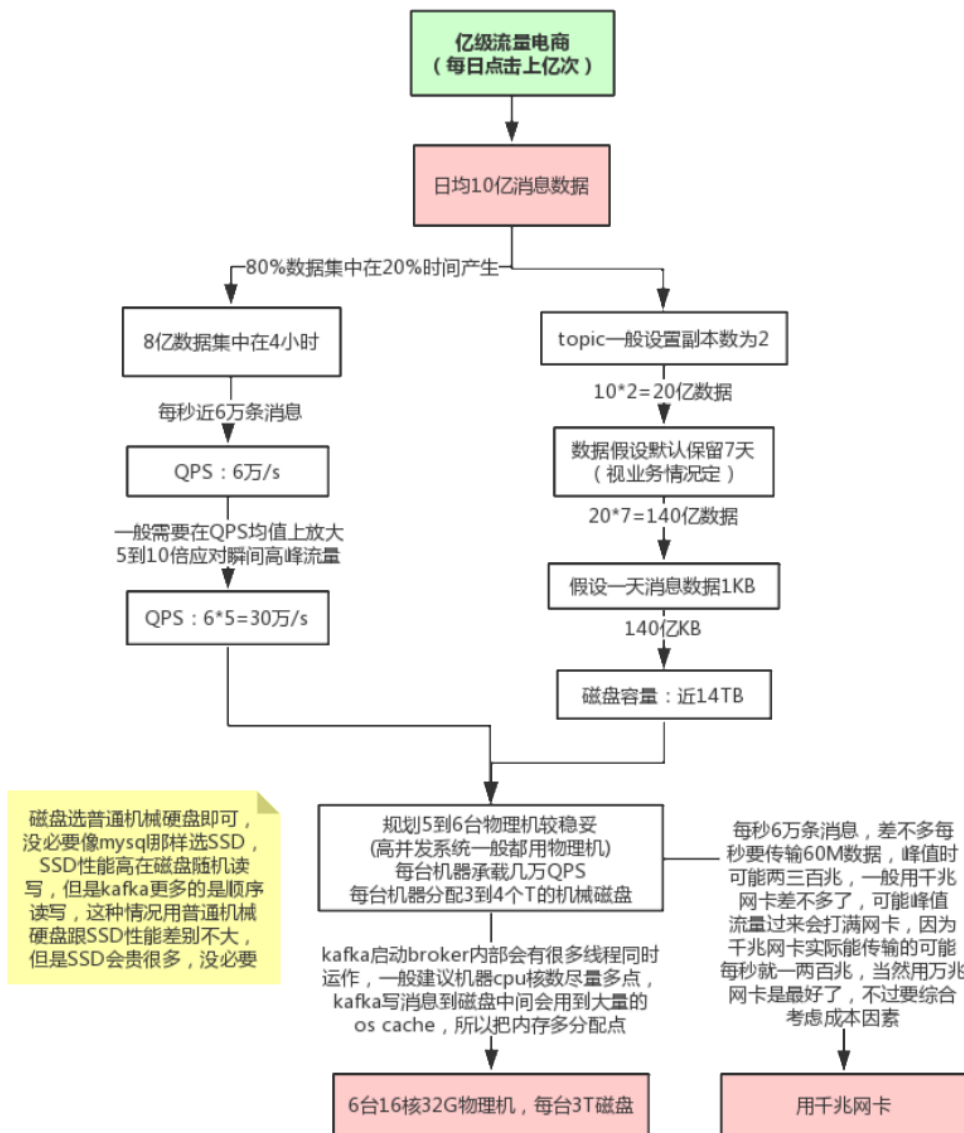
Topic	# Partitions	# Brokers	Brokers Spread %	Brokers Skew %	Brokers Leader Skew %	# Replicas	Under Replicated %
__consumer_offsets	50	3	100	0	0	1	0
my-replicated-topic	2	3	100	0	0	3	0
mytopic	1	1	33	0	0	1	0
order-topic	4	3	100	0	0	1	0
test	1	1	33	0	0	1	0
zhuge-1	2	3	100	0	0	3	0

Showing 1 to 6 of 6 entries

Previous1Next

安装及基本使用可参考：<https://www.cnblogs.com/dadonggg/p/8205302.html>

线上环境规划



JVM参数设置

kafka是scala语言开发，运行在JVM上，需要对JVM参数合理设置，参看JVM调优专题

修改bin/kafka-start-server.sh中的jvm设置，假设机器是32G内存，可以如下设置：

```
1 export KAFKA_HEAP_OPTS="-Xmx16G -Xms16G -Xmn10G -XX:MetaspaceSize=256M -XX:+UseG1GC -XX:MaxGCPauseMillis=50 -XX:G1HeapRegionSize=16M"
```

这种大内存的情况一般都要用G1垃圾收集器，因为年轻代内存比较大，用G1可以设置GC最大停顿时间，不至于一次minor gc就花费太长时间，当然，因为像kafka，rocketmq，es这些中间件，写数据到磁盘会用到操作系统的page cache，所以JVM内存不宜分配过大，需要给操作系统的缓存留出几个G。

线上问题及优化

1、消息丢失情况：

消息发送端：

- (1) acks=0：表示producer不需要等待任何broker确认收到消息的回复，就可以继续发送下一条消息。性能最高，但是最容易丢消息。大数据统计报表场景，对性能要求很高，对数据丢失不敏感的情况可以用这种。
- (2) acks=1：至少要等待leader已经成功将数据写入本地log，但是不需要等待所有follower是否成功写入。就可以继续发送下一条消息。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=-1或all：这意味着leader需要等待所有备份(min.insync.replicas配置的备份个数)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。这是最强的数据保证。一般除非是金融级别，或跟钱打交道的场景才会使用这种配置。当然如果min.insync.replicas配置的是1则也可能丢消息，跟acks=1情况类似。

消息消费端：

如果消费这边配置的是自动提交，万一消费到数据还没处理完，就自动提交offset了，但是此时你consumer直接宕机了，未处理完的数据丢失了，下次也消费不到了。

2、消息重复消费

消息发送端：

发送消息如果配置了重试机制，比如网络抖动时间过长导致发送端发送超时，实际broker可能已经接收到消息，但发送方会重新发送消息

消息消费端：

如果消费这边配置的是自动提交，刚拉取了一批数据处理了一部分，但还没来得及提交，服务挂了，下次重启又会拉取相同的一批数据重复处理

一般消费端都是要做消费幂等处理的。

3、消息乱序

如果发送端配置了重试机制，kafka不会等之前那条消息完全发送成功才去发送下一条消息，这样可能会出现，发送了1，2，3条消息，第一条超时了，后面两条发送成功，再重试发送第1条消息，这时消息在broker端的顺序就是2，3，1了

所以，是否一定要配置重试要根据业务情况而定。也可以用同步发送的模式去发消息，当然acks不能设置为0，这样也能保证消息从发送端到消费端全链路有序。

kafka保证全链路消息顺序消费，需要从发送端开始，将所有有序消息发送到同一个分区，然后用一个消费者去消费，但是这种性能比较低，可以在消费者端接收到消息后将需要保证顺序消费的几条消费发到内存队列(可以搞多个)，一个内存队列开启一个线程顺序处理消息。

4、消息积压

1) 线上有时因为发送方发送消息速度过快，或者消费方处理消息过慢，可能会导致broker积压大量未消费消息。

此种情况如果积压了上百万未消费消息需要紧急处理，可以修改消费端程序，让其将收到的消息快速转发到其他topic(可以设置很多分区)，然后再启动多个消费者同时消费新主题的不同分区。

2) 由于消息数据格式变动或消费者程序有bug，导致消费者一直消费不成功，也可能导致broker积压大量未消费消息。

此种情况可以将这些消费不成功的消息转发到其它队列里去(类似死信队列)，后面再慢慢分析死信队列里的消息处理问题。

5、延时队列

延时队列存储的对象是延时消息。所谓的“延时消息”是指消息被发送以后，并不想让消费者立刻获取，而是等待特定的时间后，消费者才能获取这个消息进行消费，延时队列的使用场景有很多，比如：

1) 在订单系统中，一个用户下单之后通常有 30 分钟的时间进行支付，如果 30 分钟之内没有支付成功，那么这个订单将进行异常处理，这时就可以使用延时队列来处理这些订单了。

2) 订单完成1小时后通知用户进行评价。

实现思路：发送延时消息时先把消息按照不同的延迟时间段发送到指定的队列中 (topic_1s, topic_5s, topic_10s, ...topic_2h, 这个一般不能支持任意时间段的延时)，然后通过定时器进行轮训消费这些topic，查看消息是否到期，如果到期就把这个消息发送到具体业务处理的topic中，队列中消息越靠前的到期时间越早，具体来说就是定时器在一次消费过程中，对消息的发送时间做判断，看下是否延迟到对应时间了，如果到了就转发，如果还没到这一次定时任务就可以提前结束了。

6、消息回溯

如果某段时间对已消费消息计算的结果觉得有问题，可能是由于程序bug导致的计算错误，当程序bug修复后，这时可能需要对之前已消费的消息重新消费，可以指定从多久之前的消息回溯消费，这种可以用consumer的offsetsForTimes、seek等方法指定从某个offset偏移的消息开始消费，参见上节课的内容。

7、分区数越多吞吐量越高吗

可以用kafka压测工具自己测试分区数不同，各种情况下的吞吐量

```
1 # 往test里发送一百万消息，每条设置1KB
2 # throughput 用来进行限流控制，当设定的值小于 0 时不限流，当设定的值大于 0 时，当发送的吞吐量大于该值时就会被阻塞一段时间
3 bin/kafka-producer-perf-test.sh --topic test --num-records 1000000 --record-size 1024 --throughput -1
4 --producer-props bootstrap.servers=192.168.65.60:9092 acks=1
```

```
[root@localhost kafka-2.11-1.1.0]# bin/kafka-producer-perf-test.sh --topic test --num-records 1000000 --record-size 1024 --throughput -1 --produce
tstrap.servers=192.168.0.60:9092 acks=1
17956 records sent, 3590.5 records/sec (3.51 MB/sec), 1955.0 ms avg latency, 3805.0 max latency.
23235 records sent, 4634.9 records/sec (4.53 MB/sec), 5962.8 ms avg latency, 7633.0 max latency.
36375 records sent, 7190.2 records/sec (7.02 MB/sec), 5950.0 ms avg latency, 7766.0 max latency.
70125 records sent, 14025.0 records/sec (13.70 MB/sec), 2624.7 ms avg latency, 3938.0 max latency.
79620 records sent, 15807.0 records/sec (15.44 MB/sec), 1865.3 ms avg latency, 2296.0 max latency.
54390 records sent, 10873.7 records/sec (10.62 MB/sec), 2713.4 ms avg latency, 3143.0 max latency.
59835 records sent, 11959.8 records/sec (11.68 MB/sec), 2637.4 ms avg latency, 3159.0 max latency.
78135 records sent, 15627.0 records/sec (15.26 MB/sec), 2136.0 ms avg latency, 2887.0 max latency.
63525 records sent, 12705.0 records/sec (12.41 MB/sec), 2227.3 ms avg latency, 2554.0 max latency.
51690 records sent, 10338.0 records/sec (10.10 MB/sec), 2852.1 ms avg latency, 3516.0 max latency.
71625 records sent, 14085.5 records/sec (13.76 MB/sec), 2364.2 ms avg latency, 3594.0 max latency.
34005 records sent, 6791.5 records/sec (6.63 MB/sec), 3352.6 ms avg latency, 4727.0 max latency.
19365 records sent, 3846.8 records/sec (3.76 MB/sec), 5442.7 ms avg latency, 6924.0 max latency.
54255 records sent, 10846.7 records/sec (10.59 MB/sec), 4583.1 ms avg latency, 7676.0 max latency.
101250 records sent, 20237.9 records/sec (19.76 MB/sec), 1562.6 ms avg latency, 2066.0 max latency.
124680 records sent, 24836.7 records/sec (24.25 MB/sec), 1302.9 ms avg latency, 1756.0 max latency.
1000000 records sent, 11940.868818 records/sec (11.66 MB/sec), 2513.48 ms avg latency, 7766.00 ms max latency, 2196 ms 50th, 6272 ms 95th, 7279 ms
ms 99.9th.
```

网络上很多资料都说分区数越多吞吐量越高，但从压测结果来看，分区数到达某个值吞吐量反而开始下降，实际上很多事情都会有一个临界值，当超过

这个临界值之后，很多原本符合既定逻辑的走向又会变得不同。一般情况分区数跟集群机器数量相当就差不多了。

当然吞吐量的数值和走势还会和磁盘、文件系统、I/O调度策略等因素相关。

注意：如果分区数设置过大，比如设置10000，可能会设置不成功，后台会报错"java.io.IOException : Too many open files"。

异常中最关键的信息是 "Too many open files"，这是一种常见的 Linux 系统错误，通常意味着文件描述符不足，它一般发生在创建线程、创建 Socket、打开文件这些场景下。在 Linux 系统的默认设置下，这个文件描述符的个数不是很多，通过 `ulimit -n` 命令可以查看：一般默认是1024，可以将该值增大，比如：`ulimit -n 65535`

8、消息传递保障

- at most once(消费者最多收到一次消息，0-1次): `acks = 0` 可以实现。
- at least once(消费者至少收到一次消息，1-多次): `ack = all` 可以实现。
- exactly once(消费者刚好收到一次消息): at least once 加上消费者幂等性可以实现，还可以用kafka生产者的幂等性来实现。

kafka生产者的幂等性：因为发送端重试导致的消息重复发送问题，kafka的幂等性可以保证重复发送的消息只接收一次，只需在生产者加上参数 `props.put("enable.idempotence", true)` 即可，默认是false不开启。

具体实现原理是，kafka每次发送消息会生成PID和Sequence Number，并将这两个属性一起发送给broker，broker会将PID和Sequence Number跟消息绑定一起存起来，下次如果生产者重发相同消息，broker会检查PID和Sequence Number，如果相同不会再接收。

- 1 **PID**：每个新的 Producer 在初始化的时候会被分配一个唯一的 **PID**，这个**PID** 对用户完全是透明的。生产者如果重启则会生成新的**PID**。
- 2 **Sequence Number**：对于每个 **PID**，该 Producer 发送到每个 Partition 的数据都有对应的序列号，这些序列号是从0开始单调递增的。

9、kafka的事务

Kafka的事务不同于Rocketmq，Rocketmq是保障本地事务(比如数据库)与mq消息发送的事务一致性，Kafka的事务主要是保障一次发送多条消息的事务一致性(要么同时成功要么同时失败)，一般在kafka的流式计算场景用得更多一点，比如，kafka需要对一个topic里的消息做不同的流式计算处理，处理完分别发到不同的topic里，这些topic分别被不同的下游系统消费(比如hbase, redis, es等)，这种我们肯定希望系统发送到多个topic的数据保持事务一致性。Kafka要实现类似Rocketmq的分布式事务需要额外开发功能。

kafka的事务处理可以参考[官方文档](#)：

```
1 Properties props = new Properties();
2 props.put("bootstrap.servers", "localhost:9092");
3 props.put("transactional.id", "my-transactional-id");
4 Producer<String, String> producer = new KafkaProducer<>(props, new StringSerializer(), new StringSerializer());
5 //初始化事务
6 producer.initTransactions();
7
8 try {
9 //开启事务
10 producer.beginTransaction();
11 for (int i = 0; i < 100; i++){
12 //发到不同的主题的不同分区
13 producer.send(new ProducerRecord<>("hdfs-topic", Integer.toString(i), Integer.toString(i)));
14 producer.send(new ProducerRecord<>("es-topic", Integer.toString(i), Integer.toString(i)));
15 producer.send(new ProducerRecord<>("redis-topic", Integer.toString(i), Integer.toString(i)));
16 }
17 //提交事务
```

```

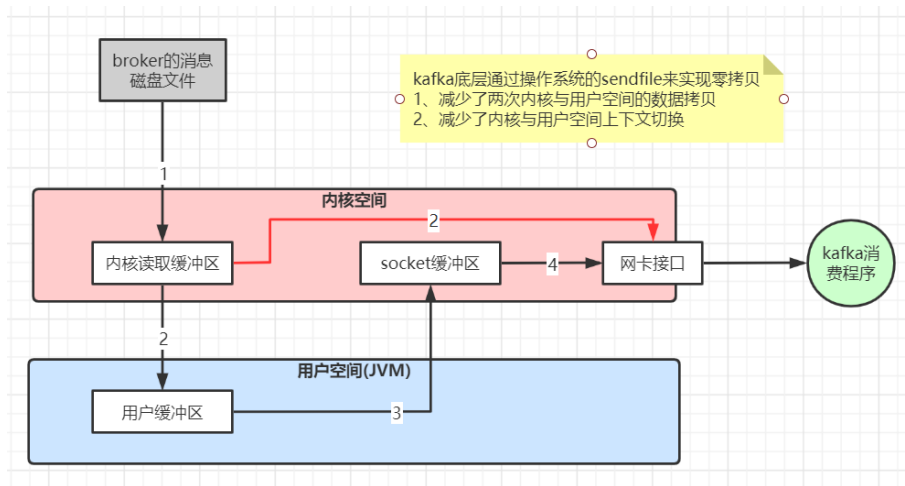
18 producer.commitTransaction();
19 } catch (ProducerFencedException | OutOfOrderSequenceException | AuthorizationException e) {
20 // We can't recover from these exceptions, so our only option is to close the producer and exit.
21 producer.close();
22 } catch (KafkaException e) {
23 // For all other exceptions, just abort the transaction and try again.
24 //回滚事务
25 producer.abortTransaction();
26 }
27 producer.close();

```

10、kafka高性能的原因

- 磁盘顺序读写：kafka消息不能修改以及不会从文件中间删除保证了磁盘顺序读，kafka的消息写入文件都是追加在文件末尾，不会写入文件中的某个位置(随机写)保证了磁盘顺序写。
- 数据传输的零拷贝
- 读写数据的批量batch处理以及压缩传输

数据传输零拷贝原理：



- 1 有道云笔记：
- 2 文档： [03-VIP-Kafka性能优化最佳实践.note](#)
- 3 链接： <http://note.youdao.com/noteshare?id=1ce32fe44f9326456e98b6e554bf6245&sub=EE921DB4564C46909157C934239C3D22>