kafka常见问题

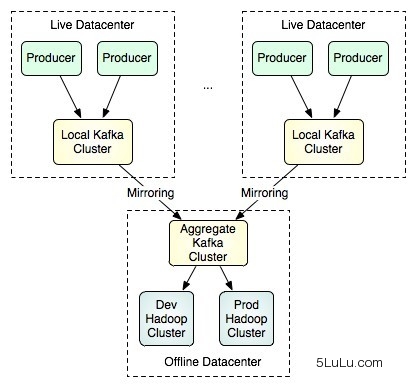
## 1、如果想消费已经被消费过的数据

consumer是底层采用的是一个阻塞队列，只要一有producer生产数据，那consumer就会将数据消费。当然这里会产生一个很严重的问题，如果你重启一消费者程序，那你连一条数据都抓不到，但是log文件中明明可以看到所有数据都好好的存在。换句话说，一旦你消费过这些数据，那你就无法再次用同一个groupid消费同一组数据了。

原因：消费者消费了数据并不从队列中移除，只是记录了offset偏移量。同一个consumergroup的所有consumer合起来消费一个topic，并且他们每次消费的时候都会保存一个offset参数在zookeeper的root上。如果此时某个consumer挂了或者新增一个consumer进程，将会触发kafka的负载均衡，暂时性的重启所有consumer，重新分配哪个consumer去消费哪个partition，然后再继续通过保存在zookeeper上的offset参数继续读取数据。注意:offset保存的是consumer 组消费的消息偏移。

要消费同一组数据，你可以

1. 采用不同的group。
2. 通过一些配置，就可以将线上产生的数据同步到镜像中去，然后再由特定的集群区处理大批量的数据。



## 2、如何自定义去消费已经消费过的数据

Conosumer.properties配置文件中有两个重要参数

auto.commit.enable：如果为true，则consumer的消费偏移offset会被记录到zookeeper。下次consumer启动时会从此位置继续消费。

auto.offset.reset 该参数只接受两个常量largest和Smallest,分别表示将当前offset指到日志文件的最开始位置和最近的位置。

如果进一步想控制时间，则需要调用SimpleConsumer，自己去设置相关参数。比较重要的参数是 kafka.api.OffsetRequest.EarliestTime()和kafka.api.OffsetRequest.LatestTime()分别表示从日志（数据）的开始位置读取和只读取最新日志。

如何使用SimpleConsumer

首先，你必须知道读哪个topic的哪个partition

然后，找到负责该partition的broker leader，从而找到存有该partition副本的那个broker

再者，自己去写request并fetch数据

最终，还要注意需要识别和处理brokerleader的改变

## 3、kafka partition和consumer数目关系

1. 如果consumer比partition多，是浪费，因为kafka的设计是在一个partition上是不允许并发的，所以consumer数不要大于partition数 。
2. 如果consumer比partition少，一个consumer会对应于多个partitions，这里主要合理分配consumer数和partition数，否则会导致partition里面的数据被取的不均匀 。最好partiton数目是consumer数目的整数倍，所以partition数目很重要，比如取24，就很容易设定consumer数目 。
3. 如果consumer从多个partition读到数据，不保证数据间的顺序性，kafka只保证在一个partition上数据是有序的，但多个partition，根据你读的顺序会有不同
4. 增减consumer，broker，partition会导致rebalance，所以rebalance后consumer对应的partition会发生变化

## 4、kafka topic 副本问题

Kafka尽量将所有的Partition均匀分配到整个集群上。一个典型的部署方式是一个Topic的Partition数量大于Broker的数量。

1. 如何分配副本:

Producer在发布消息到某个Partition时，先通过ZooKeeper找到该Partition的Leader，然后无论该Topic的Replication Factor为多少（也即该Partition有多少个Replica），Producer只将该消息发送到该Partition的Leader。Leader会将该消息写入其本地Log。每个Follower都从Leader pull数据。这种方式上，Follower存储的数据顺序与Leader保持一致。

1. Kafka分配Replica的算法如下：

将所有Broker（假设共n个Broker）和待分配的Partition排序

将第i个Partition分配到第（imod n）个Broker上

将第i个Partition的第j个Replica分配到第（(i + j) mode n）个Broker上

## 5、kafka如何设置生存周期与清理数据

日志文件的删除策略非常简单:启动一个后台线程定期扫描log file列表,把保存时间超过阈值的文件直接删除(根据文件的创建时间).清理参数在server.properties文件中：



## 6、zookeeper如何管理kafka

1. Producer端使用zookeeper用来"发现"broker列表,以及和Topic下每个partition leader建立socket连接并发送消息.
2. Broker端使用zookeeper用来注册broker信息,以及监测partition leader存活性.
3. Consumer端使用zookeeper用来注册consumer信息,其中包括consumer消费的partition列表等,同时也用来发现broker列表,并和partition leader建立socket连接,并获取消息.

## 7、SparkStreaming之Kafka的Receiver和Direct方式讲解

### 7.1、Receiver方式

Receiver是使用Kafka的high level的consumer API来实现的。Receiver从Kafka中获取数据都是存储在Spark Executor内存中的，然后Spark Streaming启动的job会去处理那些数据

然而这种方式很可能会丢失数据，如果要启用高可靠机制，让数据零丢失，就必须启动Spark Streaming预写日志（wal）机制。该机制会同步地接收到Kafka数据写入分布式文件系统，比如HDFS上的预写日志中。所以底层节点出现了失败，也可以使用预写日志的数据进行恢复。

### 7.2、Direct直连方式

它会周期性的查询kafka，来获取每个topic + partition的最新offset，从而定义每一个batch的offset的范围。当处理数据的job启动时，就会使用kafka简单的消费者API来获取kafka指定offset的范围的数据。

1）它简化了并行读取：如果要读取多个partition，不需要创建多个输入DStream然后对他们进行union操作。Spark会创建跟kafka partition一样多的RDD partition，并且会并行从kafka中读取数据。所以在kafka partition和RDD partition之间有一一对应的映射关系。

2）高性能：如果要保证数据零丢失，基于Receiver的机制需要开启WAL机制，这种方式其实很低效，因为数据实际上被copy了2分，kafka自己本身就有可靠的机制，会对数据复制一份，而这里又复制一份到WAL中。基于Direct的方式，不依赖于Receiver，不需要开启WAL机制,只要kafka中做了数据的复制，那么就可以通过kafka的副本进行恢复。

3）一次仅且一次的事务机制

基于Receiver的方式，是使用Kafka High Level的API在zookeeper中保存消费过的offset的。这是消费kafka数据的传统方式，这种方式配合这WAL机制可以保证数据零丢失，但是无法保证数据只被处理一次的且仅且一次，可能会两次或者更多，因为spark和zookeeper可能是不同步的。

4）降低资源

Direct不需要Receivers，其申请的Executors全部参与到计算任务中；而Receiver-based则需要专门的Receivers来读取Kafka数据且不参与计算。因此相同的资源申请，Direct 能够支持更大的业务。

5）降低内存

Receiver-based的Receiver与其他Exectuor是异步的，并持续不断接收数据，对于小业务量的场景还好，如果遇到大业务量时，需要提高Receiver的内存，但是参与计算的Executor并无需那么多的内存。而Direct 因为没有Receiver，而是在计算时读取数据，然后直接计算，所以对内存的要求很低。实际应用中我们可以把原先的10G降至现在的2-4G左右。

6）不会出现数据堆积

Receiver-based方法需要Receivers来异步持续不断的读取数据，因此遇到网络、存储负载等因素，导致实时任务出现堆积，但Receivers却还在持续读取数据，此种情况很容易导致计算崩溃。Direct 则没有这种顾虑，其Driver在触发batch 计算任务时，才会读取数据并计算。队列出现堆积并不会引起程序的失败。

基于direct的方式，使用kafka的简单api，Spark Streaming自己就负责追踪消费的offset，并保存在checkpoint中。Spark自己一定是同步的，因此可以保证数据是消费一次且仅消费一次。