Kafka介绍

2017年11月9日 10:11

概述

官方网址：http://kafka.apache.org/

以下摘自官网的介绍：

Apache Kafka® is *a distributed streaming platform*. What exactly does that mean?

We think of a streaming platform as having three key capabilities:

1. It lets you publish and subscribe to streams of records. In this respect it is similar to a **message queue** or enterprise messaging system.
2. It lets you **store** streams of records in a **fault-tolerant** way.
3. It lets you process streams of records as they occur.

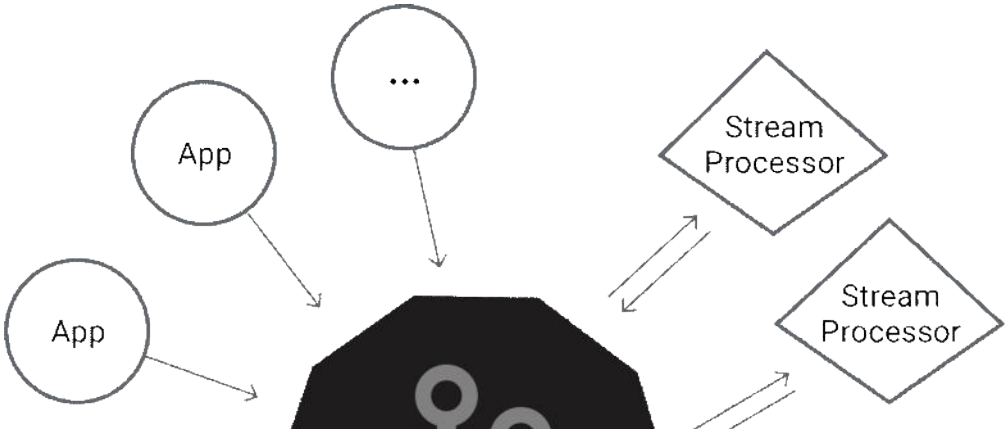
What is Kafka good for?

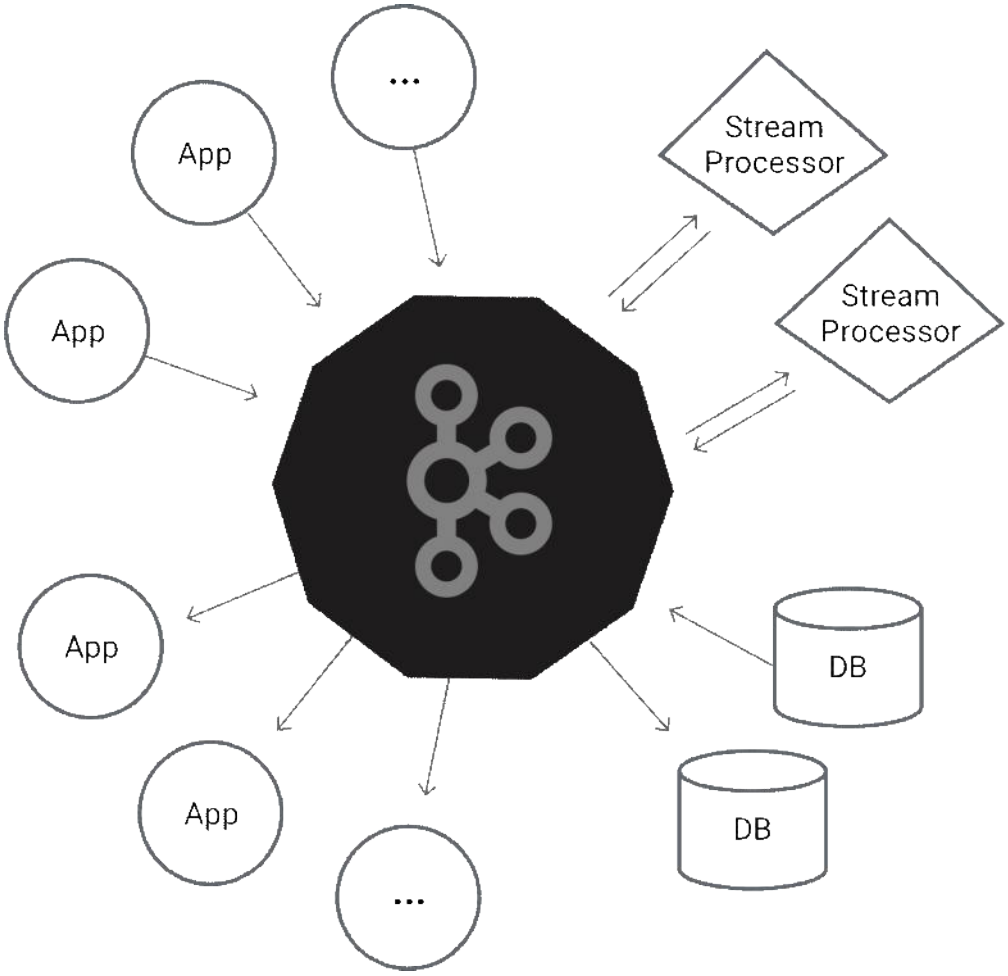
It gets used for two broad classes of application:

1. Building **real-time streaming** data pipelines that reliably **get data** between systems or applications
2. Building **real-time streaming** applications that **transform** or **react** to the streams of data

First a few concepts:

- Kafka is run as a cluster on one or more servers.
- The Kafka cluster stores streams of *records* in categories called ***topics***.
- Each record consists of a key, a value, and a timestamp.





Kafka是由LinkedIn开发的一个**分布式**的消息系统，最初是用作LinkedIn的**活动流**（Activity Stream）和**运营数据**处理的基础。

活动流数据包括页面访问量（Page View）、被查看内容方面的信息以及搜索情况等内容。这种数据通常的处理方式是先把各种活动以日志的形式写入某种文件，然后周期性地对这些文件进行统计分析。

运营数据指的是服务器的性能数据（CPU、IO使用率、请求时间、服务日志等等数据)。运营数据的统计方法种类繁多。

Kafka使用Scala编写，它以可水平扩展和高吞吐率而被广泛使用。目前越来越多的开源分布

式处理系统如Cloudera、Apache Storm、Spark都支持与Kafka集成。

综上，Kafka是一种分布式的，基于发布/订阅的消息系统，能够高效并实时的**吞吐**数据，以及通过分布式集群及数据复制冗余机制（副本冗余机制）实现数据的安全

常用Message Queue对比

RabbitMQ

RabbitMQ是使用Erlang编写的一个开源的消息队列，本身支持很多的协议：AMQP，XMPP, SMTP, STOMP，也正因如此，它非常重量级，更适合于企业级的开发。同时实现了Broker构架，这意味着消息在发送给客户端时先在中心队列排队。对路由，负载均衡或者数据持久化都有很好的支持。

Redis

Redis是一个基于Key-Value对的NoSQL数据库，开发维护很活跃。虽然它是一个Key-Value数据库存储系统，但它本身支持MQ功能，所以完全可以当做一个轻量级的队列服务来使用。

ZeroMQ

ZeroMQ号称最快的消息队列系统，尤其针对大吞吐量的需求场景。ZeroMQ能够实现RabbitMQ不擅长的高级/复杂的队列，但是开发人员需要自己组合多种技术框架，技术上的复杂度是对这MQ能够应用成功的挑战。但是ZeroMQ仅提供非持久性的队列，也就是说如果宕机，数据将会丢失。其中，Twitter的Storm 0.9.0以前的版本中默认使用ZeroMQ作为数据流的传输（Storm从0.9版本开始同时支持ZeroMQ和Netty（NIO）作为传输模块）。

ActiveMQ

ActiveMQ是Apache下的一个子项目。 类似于ZeroMQ，它能够以代理人和点对点的技术实

现队列。同时类似于RabbitMQ，它少量代码就可以高效地实现高级应用场景。

适用场景

Messaging

对于一些常规的消息系统,kafka是个不错的选择;partitons/replication和容错,可以使kafka具有良好的扩展性和性能优势.不过到目前为止,我们应该很清楚认识到,kafka并没有提供JMS中的"事务性""消息传输担保(消息确认机制)""消息分组"等企业级特性;kafka只能使用作为"常规"的消息系统,在一定程度上,尚未确保消息的发送与接收绝对可靠(比如,消息重发,消息发送丢失等)

Website activity tracking

kafka可以作为"网站活性跟踪"的最佳工具;可以将网页/用户操作等信息发送到kafka中.并实时监控,或者离线统计分析等

Metric

Kafka通常被用于可操作的监控数据。这包括从分布式应用程序来的聚合统计用来生产集中的运营数据提要。

Log Aggregatio

kafka的特性决定它非常适合作为"日志收集中心";application可以将操作日志"批量""异步"的发送到kafka集群中,而不是保存在本地或者DB中;kafka可以批量提交消息/压缩消息等,这对producer端而言,几乎感觉不到性能的开支.此时consumer端可以使hadoop等其他系统化的存储和分析系统

Kafka配置

2017年11月16日 15:09

实现步骤：

1.从官网下载安装包 http://kafka.apache.org/downloads

2.上传到01虚拟机，解压

3.进入安装目录下的config目录

4.对server.properties进行配置

配置示例：

broker.id=0

log.dirs=/home/software/kafka/kafka-logs

zookeeper.connect=hadoop01:2181,hadoop02:2181,hadoop03:2181

delete.topic.enable=true

advertised.host.name=192.168.234.21

advertised.port=9092

5.保存退出后，别忘了在安装目录下创建 kafka-logs目录

6.配置其他两台虚拟机，更改配置文件的broker.id编号（不重复即可）并修改 advertised.host.name ip 地址为本机虚拟机地址

7.先启动zookeeper集群

8.启动kafka集群

进入bin目录

执行：sh kafka-server-start.sh ../config/server.properties

Kafka在Zookeeper下的路径：

```
rmr /cluster
rmr /brokers
rmr /admin
rmr /isr_change_notification
rmr /log_dir_event_notification
rmr /controller_epoch

rmr /consumers
rmr /latest_producer_id_block
rmr /config
```

Kafka使用

1.创建自定义的topic
在bin目录下执行：
sh kafka-topics.sh --create --zookeeper hadoop01:2181 --replication-factor 1 --partitions 1 --topic enbook

注:副本数量要小于等于节点数量

2.查看所有的topic
执行：sh kafka-topics.sh --list --zookeeper hadoop01:2181

3.启动producer
执行：sh kafka-console-producer.sh --broker-list hadoop01:9092,hadoop02:9092,hadoop03:9092 --topic enbook

4.启动consumer
执行：[root@hadoop01 bin]# sh kafka-console-consumer.sh --zookeeper hadoop01:2181 --topic enbook --from-beginning

5.可以通过producer和consumer模拟消息的发送和接收

6.删除topic指令：
进入bin目录，执行：sh kafka-topics.sh --delete --zookeeper hadoop01:2181 --topic enbook

可以通过配置 config目录下的 server.properties文件，加入如下的配置：

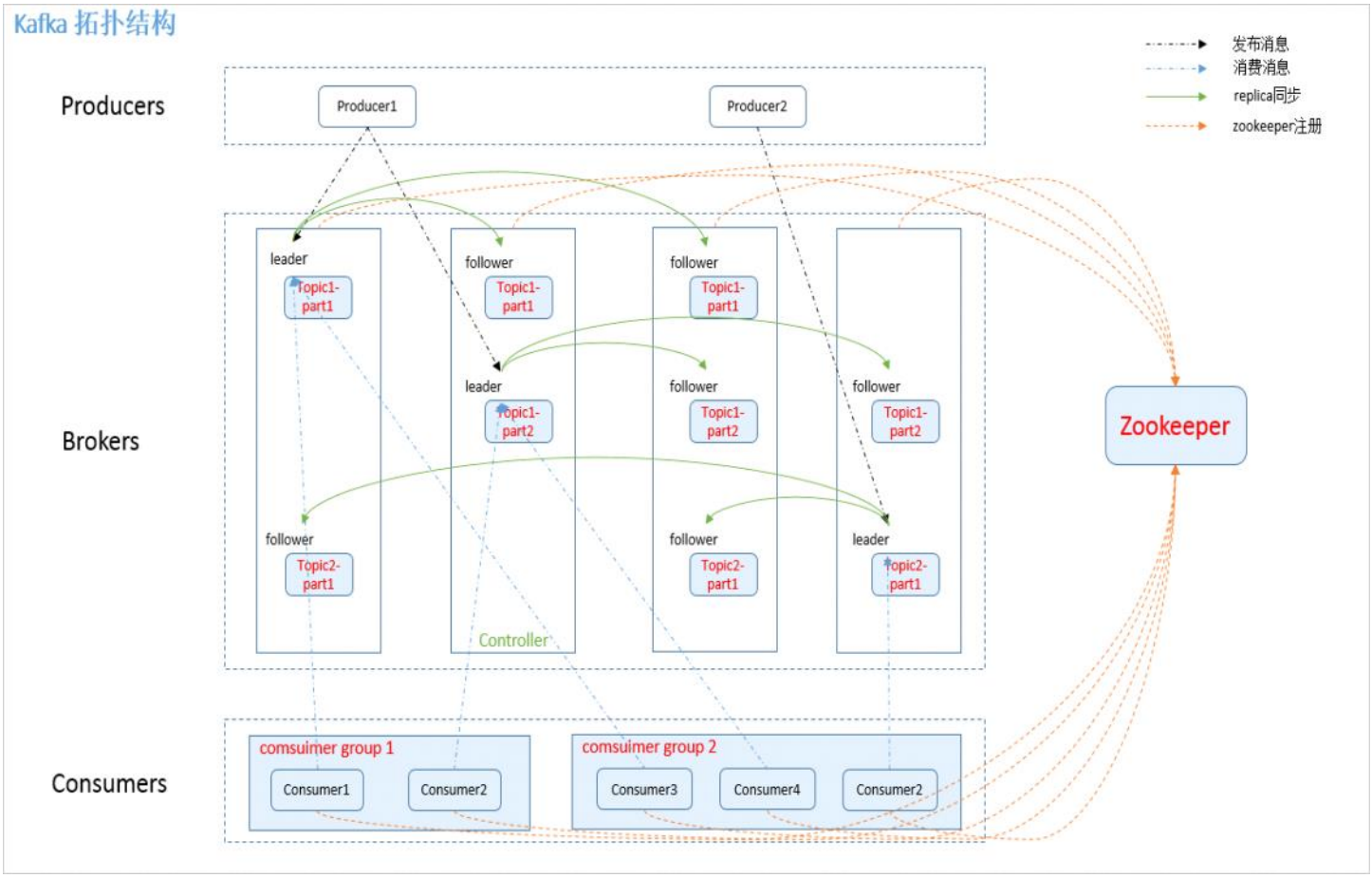
配置示例：

delete.topic.enable=true

```
advertised.host.name=hadoop
advertised.port=9092
delete.topic.enable=true
```

Kafka架构

2017年11月21日 19:14



1.producer:

消息生产者，发布消息到 kafka 集群的终端或服务。

2.broker:

kafka 集群中包含的服务器。broker (经纪人，消费转发服务)

3.topic:

每条发布到 kafka 集群的消息属于的类别，即 kafka 是面向 topic 的。

4.partition:

partition 是物理上的概念，每个 topic 包含一个或多个 partition。kafka 分配的单位是 partition。

5.consumer:

从 kafka 集群中消费消息的终端或服务。

6.Consumer group:

high-level consumer API 中，每个 consumer 都属于一个 consumer group，每条消息只能被 consumer group 中的一个 Consumer 消费，但可以被多个 consumer group 消费。

即**组间数据是共享的，组内数据是竞争的。**

7.replica:

partition 的副本，保障 partition 的高可用。

8.leader:

replica 中的一个角色， producer 和 consumer 只跟 leader 交互。

9.follower:

replica 中的一个角色，从 leader 中复制数据。

10.controller:

kafka 集群中的其中一个服务器，用来进行 leader election 以及 各种 failover。

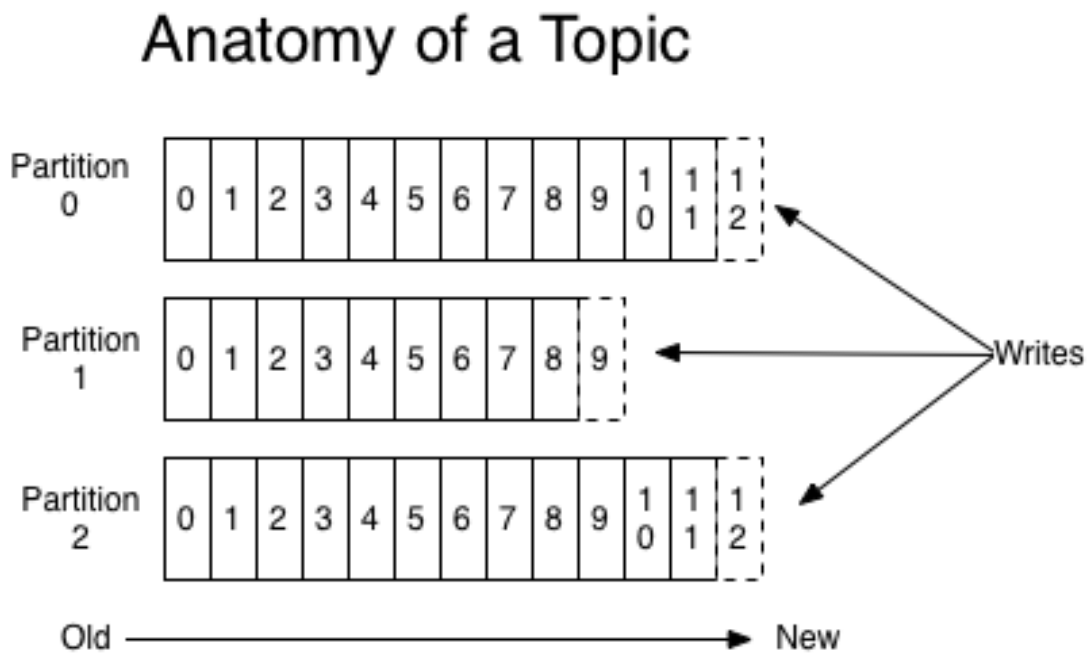
11.zookeeper:

kafka 通过 zookeeper 来存储集群的 meta 信息。

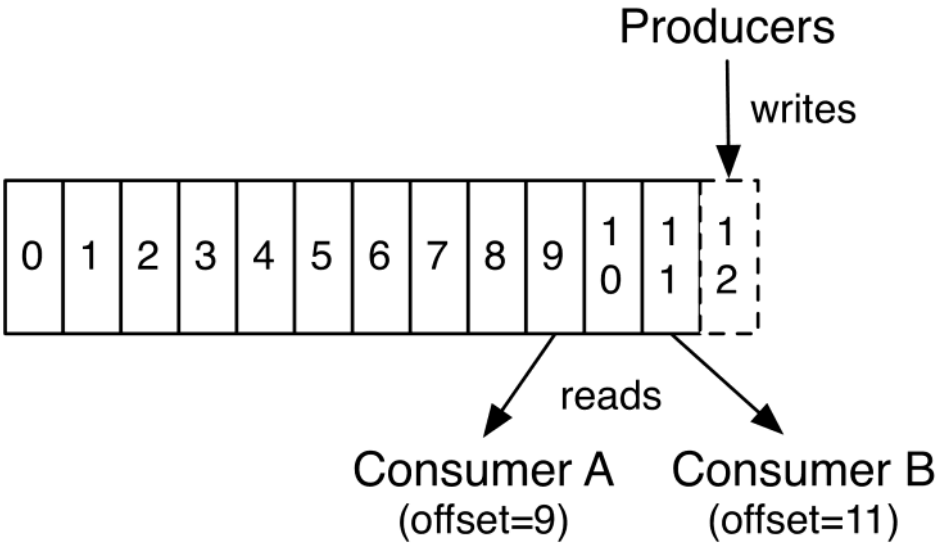
Topic与Partition

2017年11月22日 19:28

示意图



根据上图，可以看出向kakfa的主题分区写入数据时，是磁盘的顺序写，所以写入性能很高



Topic

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（主题）。

Partition

Parition是物理上的概念，每个Topic包含一个或多个Partition.

Parition从物理概念来看，对应的就是一个文件目录，目录的命名规则：主题名-分区编号（从0开始）

比如：cnbook-0 ,cnbook-1……

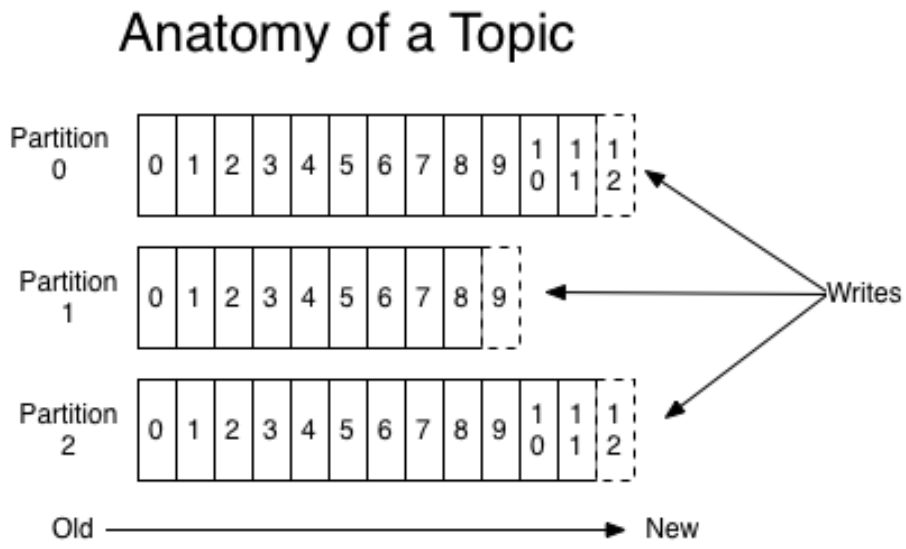
而且kafka底层对于分区目录的分配，可以达到负载均衡的效果。

Topic在逻辑上可以被认为是一个queue，每条消息都必须指定它的Topic，可以简单理解为必须指明把这条消息放进哪个queue里。

为了使得Kafka的吞吐率可以线性提高，物理上把Topic分成一个或多个Partition，**每个Partition在物理上对应一个文件夹**，该文件夹下存储这个Partition的所有消息和索引文件。若创建topic1和topic2两个topic，且分别有13个和19个分区，如下图所示。

```
node4: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic1-10
node4: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic1-2
node4: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-10
node4: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-18
node4: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-2
node2: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic1-0
node2: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic1-8
node2: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-0
node2: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-16
node2: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-8
node8: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic1-6
node8: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-14
node8: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-6
node7: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic1-5
node7: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-13
node7: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-5
node3: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic1-1
node3: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic1-9
node3: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-1
node3: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-17
node3: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-9
node6: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic1-12
node6: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic1-4
node6: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-12
node6: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-4
node5: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic1-11
node5: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic1-3
node5: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-11
node5: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-3
node1: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic1-7
node1: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-15
node1: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-7
```

因为每条消息都被append到该Partition中，属于**顺序写磁盘**，因此效率非常高（经验证，顺序写磁盘效率比随机写内存还要高，这是**Kafka高吞吐率的一个很重要的保证**）。



对于传统的message queue而言，一般会删除已经被消费的消息，**而Kafka集群会保留所有的消息，无论其被消费与否。**当然，因为磁盘限制，不可能永久保留所有数据（实际上也没必要），因此Kafka提供两种策略删除旧数据。一是基于时间，二是基于Partition文件大小。例如可以通过配置$KAFKA_HOME/config/server.properties，让Kafka删除一周前的数据，也可在Partition文件超过1GB时删除旧数据，配置如下所示。

配置示例：

```
# The minimum age of a log file to be eligible for deletion
log.retention.hours=168
# The maximum size of a log segment file. When this size is reached a new log segment will be created.
log.segment.bytes=1073741824
# The interval at which log segments are checked to see if they can be deleted according to the retention policies
log.retention.check.interval.ms=300000
```

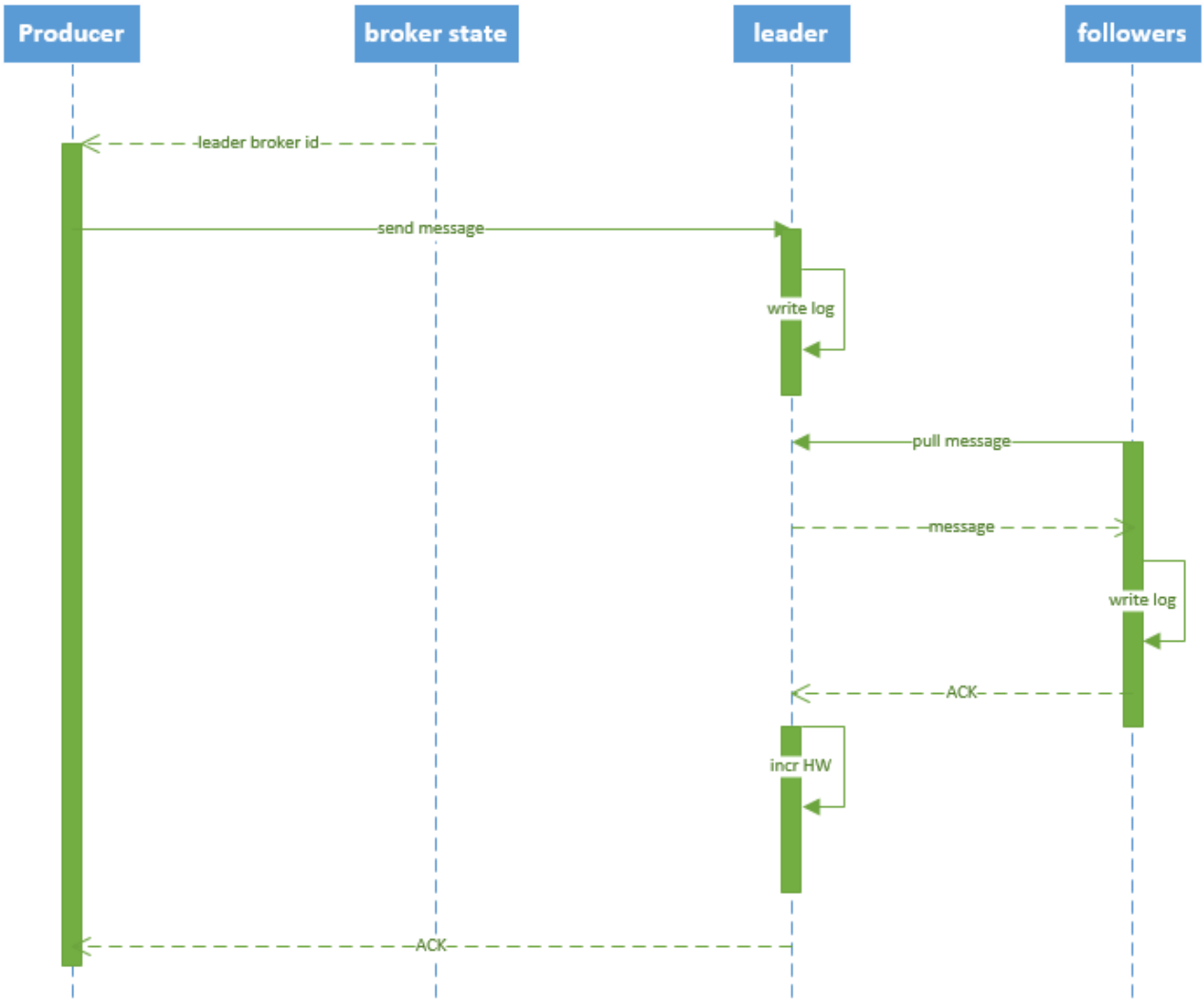
If log.cleaner.enable=true is set the cleaner will be enabled and individual logs can then be marked for log compaction.

log.cleaner.enable=false

Kafka消息流处理

2017年11月22日 19:35

Producer 写入消息序列图



流程说明：

1. producer 先从 zookeeper 的 "/brokers/.../state" 节点找到该 partition 的 leader

2. producer 将消息发送给该 leader

3. leader 将消息写入本地 log

4. followers 从 leader pull 消息，写入本地 log 后 leader 发送 ACK

5. leader 收到所有 **ISR** 中的 replica 的 ACK 后，增加 HW（high watermark，最后 commit 的 offset） 并向 producer 发送 ACK。

ISR指的是：比如有三个副本，编号是① ② ③，其中② 是Leader ① ③是Follower。假设在数据同步过程中，①跟上Leader,但是③出现故障或没有及时同步，则① ②是一个ISR，而③不是ISR成员。后期在Leader选举时，会用到ISR机制。会优先从ISR中选择Leader。因为ISR中的副本成员数据同步是一致的

kafka HA

2017年11月22日 19:47

概述

同一个 partition 可能会有多个 replica（对应 server.properties 配置中的 default.replication.factor=N）。

没有 replica 的情况下，一旦 broker 宕机，其上所有 patition 的数据都不可被消费，同时 producer 也不能再将数据存于其上的 patition。

引入replication 之后，同一个 partition 可能会有多个 replica，而这时需要在这些 replica 之间选出一个 leader，producer 和 consumer 只与这个 leader 交互，其它 replica 作为 follower 从 leader 中复制数据。

leader failover

当 partition 对应的 leader 宕机时，需要从 follower 中选举出新 leader。在选举新leader时，一个基本的原则是，新的 leader 必须拥有旧 leader commit 过的所有消息。

由写入流程可知 ISR 里面的所有 replica 都跟上了 leader，只有 ISR 里面的成员才能选为 leader。

对于 f+1 个 replica，一个 partition 可以在容忍 f 个 replica 失效的情况下保证消息不丢失。

比如 一个分区 有5个副本，挂了4个，剩一个副本，依然可以工作。

注意：kafka的选举不同于zookeeper，用的不是过半选举。

当所有 replica 都不工作时，有两种可行的方案：

1. 等待 ISR 中的任一个 replica 活过来，并选它作为 leader。可保障数据不丢失，但时间可能相对较长。

2. 选择第一个活过来的 replica（不一定是 ISR 成员）作为 leader。无法保障数据不丢失，但相对不可用时间较短。

kafka 0.8.* 使用第二种方式。此外， kafka 通过 Controller 来选举 leader。

Kafka offset机制

2018年9月6日　　20:21

Consumer消费者的offset存储机制

Consumer在从broker读取消息后，可以选择**commit**，该操作会在Kakfa中保存该Consumer在该Partition中读取的消息的**offset（位置偏移量）**。该Consumer下一次再读该Partition时会从下一条开始读取。

通过这一特性可以保证同一消费者从Kafka中不会重复消费数据。

在Kafka旧版本中，Consumer的offset是存储到Zookeeper集群上的。但是这种机制会频繁的访问zookeeper，带来过高的负载压力。因为zookeeper集群可能还要管理其他的集群（HBase,Hadoop,Dubbo等）。

所以在Kafka新版本中，针对offset机制作出更改，offset的位置偏移量是由Kafka自身来存储和管理的。新版本，Kakfa由一个特殊的主题：__consumer_offsets来存储管理的

这个主题一共有50个分区

```
__consumer_offsets-0
__consumer_offsets-1
```

……

```
__consumer_offsets-48
__consumer_offsets-49
```

底层实现原理：

执行：sh kafka-console-consumer.sh --bootstrap-server hadoop01:9092,hadoop02:9092,hadoop03:902 --topic enbook --from-beginning --new-consumer

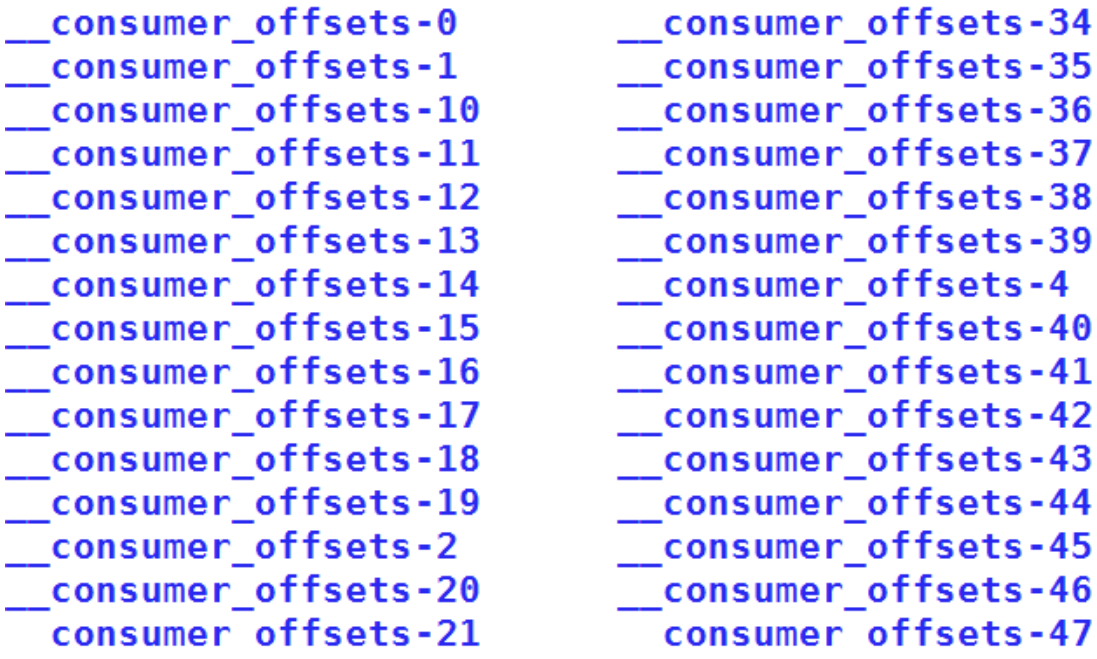
执行：sh kafka-consumer-groups.sh --bootstrap-server hadoop01:9092 --list --new-consumer

查询得到的group id

```
console-consumer-60350
```

进入kafka-logs目录查看，会发现多个很多目录，这是因为kafka默认会生成50个

__consumer_offsets 的目录，用于存储消费者消费的offset位置。



Kafka会使用下面公式计算该消费者group位移保存在__consumer_offsets的哪个目录上：

Math.abs(groupID.hashCode()) % 50

Kafka消息索引

2018年10月6日 10:18

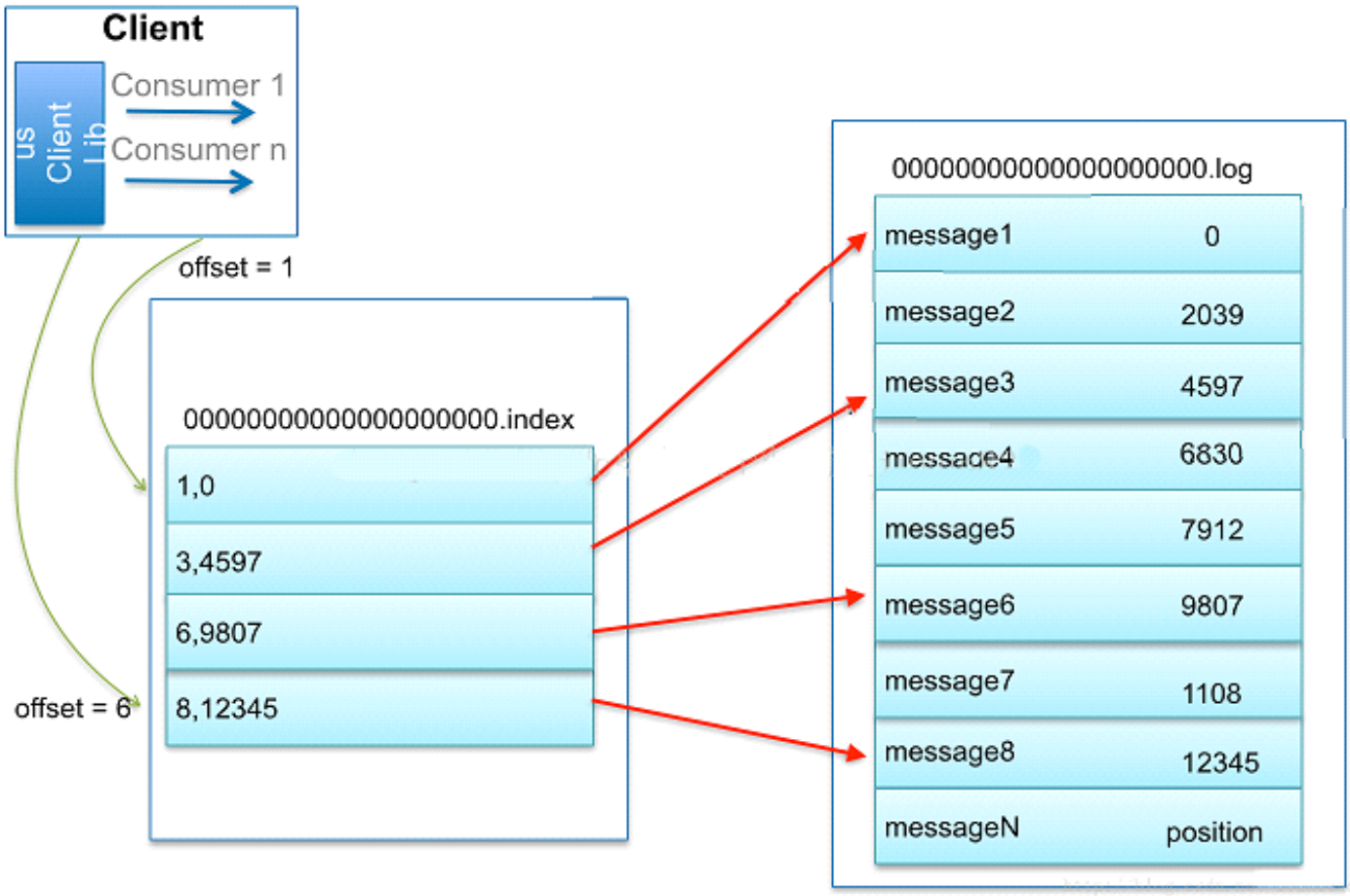
概述

数据文件的分段与索引

Kafka解决查询效率的手段之一是将数据文件分段，可以配置每个数据文件的最大值，每段放在一个单独的数据文件里面，数据文件以该段中**最小的offset命名。**

每个log文件默认是1GB生成一个新的Log文件，比如新的log文件中第一条的消息的offset 16933，则此log文件的命名为：00000000000000016933.log，此外，每生成一个log文件，就会生成一个对应的索引（index)文件。这样在查找指定offset的Message的时候，用二分查找就可以定位到该Message在哪个段中。

数据文件分段使得可以在一个较小的数据文件中查找对应offset的Message了，但是这依然需要顺序扫描才能找到对应offset的Message。为了进一步提高查找的效率，Kafka为每个分段后的数据文件建立了索引文件，文件名与数据文件的名字是一样的，只是文件扩展名为.index。索引文件中包含若干个索引条目，每个条目表示数据文件中一条Message的索引——Offset与position(Message在数据文件中的绝对位置)的对应关系。



稀疏索引+二分查找，可以加快查找速度

index文件中并没有为数据文件中的每条Message建立索引，而是采用了稀疏存储的方式，每隔一定字节的数据建立一条索引。**这样避免了索引文件占用过多的空间，从而可以将索引文件保留在内存中。**但缺点是没有建立索引的Message也不能一次定位到其在数据文件的位置，从而需要做一次顺序扫描，但是这次顺序扫描的范围就很小了。

索引文件被映射到内存中，所以查找的速度还是很快的。

扩展：Zero Copy

2017年12月5日　22:09

概述

首先来看一下维基百科对Zero Copy的定义：

（来自 <https://en.wikipedia.org/wiki/Zero-copy> ）

"**Zero-copy**" describes computer operations in which the CPU does not perform the task of copying data from one memory area to another. This is frequently used to **save CPU cycles and memory bandwidth when transmitting a file over a network.**

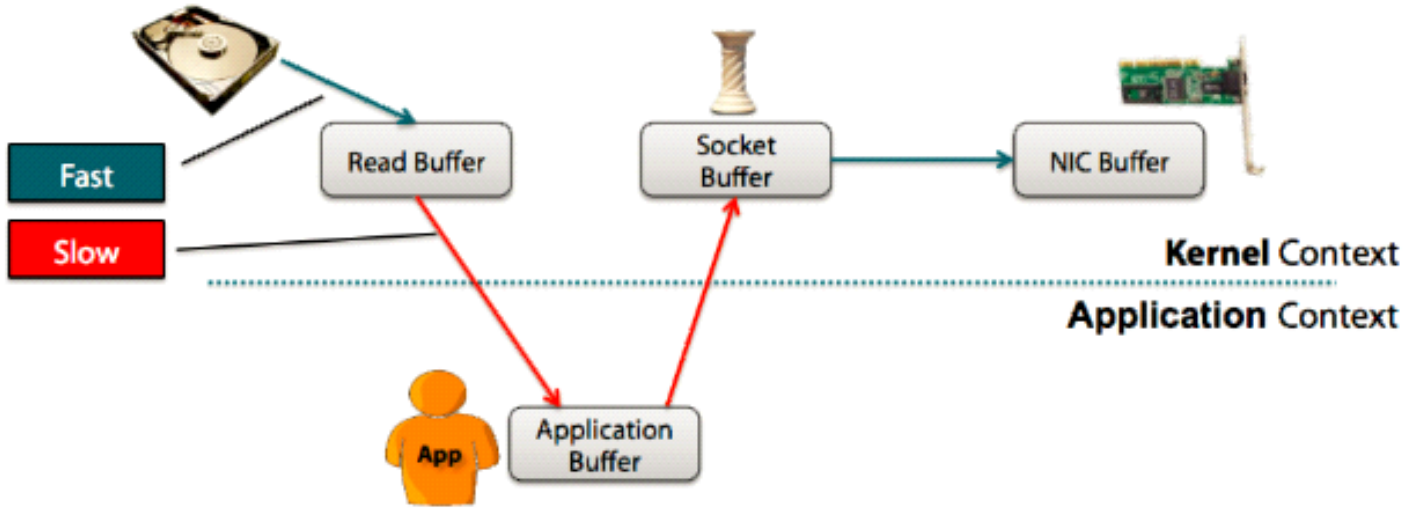
Zero-copy versions of operating system elements, such as device drivers, file systems, and network protocol stacks, greatly increase the performance of certain application programs and more efficiently utilize system resources. Also, **zero-copy operations reduce the number of time-consuming mode switches between user space and kernel space**.

原理概述

zero copy（零复制）是一种特殊形式的内存映射，它允许你将Kernel内存直接映射到设备内存空间上。其实就是设备可以通过直接内存访问（direct memory access，DMA）方式来访问Kernal Space。

我们拿Kafka举例，Kafka会对流数据做持久化存储以实现容错，比如说一个topic会对应多个Partition，每个Partition实际上最后会落地为一个文件存储在磁盘上，
这意味着当Consumer从Kafka消费数据时，会有很多data从硬盘读出之后，会原封不动的通过socket传输给用户。

这种操作看起来可能不会怎么消耗CPU，但是实际上它是低效的：kernal把数据从disk读出来，然后把它传输给user级的application，然后application再次把同样的内容再传回给处于kernal级的socket。这种场景下，application实际上只是作为一种低效的中间介质，用来把disk file的data传给socket。
如下图所示：



Non-Zero Copy

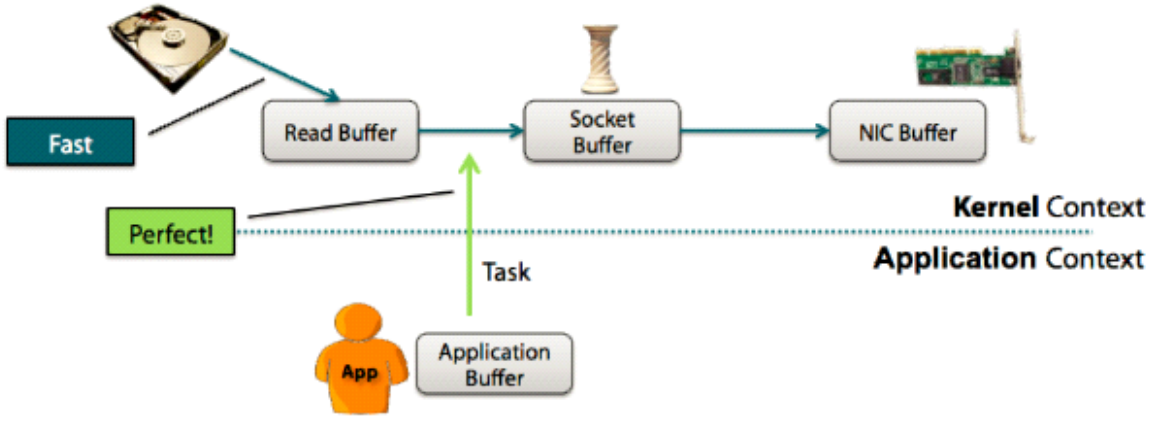
所以，当我们通过网络传输数据时，尽管看起来很简单，但是在OS的内部，这个copy操作要经历四次user mode和kernel mode之间的上下文切换，而**数据都被拷贝了四次**!

此外，data每次穿过user-kernel boundary，都会被copy，这会消耗cpu，并且占用RAM的带宽。

注：随机存取存储器(random access memory，RAM)又称作"随机存储器"，是与CPU直接交换数据的内部存储器，也叫主存(内存)。它可以随时读写，而且速度很快，通常作为操作系统或其他正在运行中的程序的临时数据存储媒介。

Zero Copy的具体实现

实际上第二次和第三次copy是毫无意义的。应用程序仅仅缓存了一下data就原封不动的把它发回给socket buffer。实际上，data应该直接在read buffer和socket buffer之间传输，如下图:



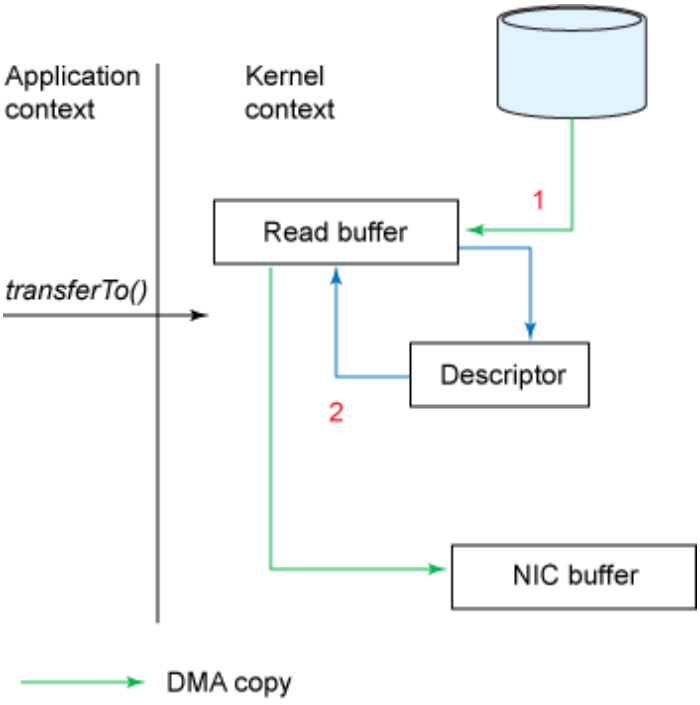
从上图中可以清楚的看到，Zero Copy的模式中，避免了数据在用户空间和内存空间之间的拷

贝，从而提高了系统的整体性能。Linux中的sendfile()以及Java NIO中的FileChannel.transferTo()方法都实现了零拷贝的功能，而在Netty中也通过在FileRegion中包装了NIO的FileChannel.transferTo()方法实现了零拷贝。

这是一个很明显的进步：我们把context switch的次数从4次减少到了2次，同时也把data copy的次数从4次降低到了3次(而且其中只有一次占用了CPU，另外两次由DMA完成)。但是，要做到真正的zero copy，这还差一些。

如果网卡支持 gather operation，我们可以通过kernel进一步减少数据的拷贝操作。在2.4及以上版本的linux内核中，开发者修改了socket buffer descriptor来适应这一需求。这个方法不仅减少了context switch，还消除了和CPU有关的数据拷贝。

如下图：



最新的Zero Copy的机制是追加了一些descriptor的信息，包括data的位置和长度。然后DMA直接把data从kernel buffer传输到protocol engine，这样就消除了唯一的一次需要占用CPU的拷贝操作。

为什么要使用kernel buffer做中介

使用kernel buffer做中介(而不是直接把data传到user buffer中)看起来比较低效(多了一次copy)。然而实际上kernel buffer是用来提高性能的。在进行读操作的时候，kernel buffer起到

了预读cache的作用。当写请求的data size比kernel buffer的size小的时候，这能够显著的提升性能。在进行写操作时，kernel buffer的存在可以使得写请求完全异步。

但悲剧的是，当读请求的data size远大于kernel buffer size的时候，这个方法本身变成了性能的瓶颈。

非重点：Kafka API使用

2017年12月5日 20:38

旧版代码示意：

```
public class TestDemo {

	@Test

	public void producer(){

		Properties props=new Properties();

		props.put("serializer.class","kafka.serializer.StringEncoder");

		props.put("metadata.broker.list","192.168.234.11:9092");

		Producer<Integer,String> producer=new Producer<>(new ProducerConfig(props));


		producer.send(new KeyedMessage<Integer, String>("enbook","Think in java"));

	}


}
```

新版代码示意：

```
import java.util.Properties;

import java.util.concurrent.ExecutionException;


import org.apache.kafka.clients.producer.KafkaProducer;

import org.apache.kafka.clients.producer.Producer;

import org.apache.kafka.clients.producer.ProducerConfig;

import org.apache.kafka.clients.producer.ProducerRecord;

import org.junit.Test;
```

```
public class TestDemo {
    @Test
    public void producer() throws InterruptedException, ExecutionException{
        Properties props=new Properties();
        props.put("key.serializer", "org.apache.kafka.common.serialization.IntegerSerializer");
        props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,"192.168.234.11:9092");

        Producer<Integer, String> kafkaProducer = new KafkaProducer<Integer, String>(props);
        for(int i=0;i<100;i++){
            ProducerRecord<Integer, String> message = new ProducerRecord<Integer, String>
            ("enbook",""+i);
            kafkaProducer.send(message);
        }

        while(true);
    }

}
```

创建Topic代码：

```
@Test
```

```
public void create_topic(){
        ZkUtils zkUtils =
        ZkUtils.apply("192.168.234.11:2181,192.168.234.210:2181,192.168.234.211:2181",
        30000, 30000, JaasUtils.isZkSecurityEnabled());
        // 创建一个单分区单副本名为t1的topic
        AdminUtils.createTopic(zkUtils, "t1", 1, 1, new Properties(), RackAwareMode.Enforced
        $.MODULE$);
        zkUtils.close();
    }
```

删除Topic代码：

```
@Test
    public void delete_topic(){
        ZkUtils zkUtils =
        ZkUtils.apply("192.168.234.11:2181,192.168.234.210:2181,192.168.234.211:2181",
        30000, 30000, JaasUtils.isZkSecurityEnabled());
        // 删除topic 't1'
        AdminUtils.deleteTopic(zkUtils, "t1");
        zkUtils.close();
    }
```

创建消费者线程并指定消费者组：

```
@Test
    public void consumer_1(){
```

```
        Properties props = new Properties();
        props.put("bootstrap.servers", "192.168.234.11:9092");
        props.put("group.id", "consumer-tutorial");
        props.put("key.deserializer", StringDeserializer.class.getName());
        props.put("value.deserializer", StringDeserializer.class.getName());
        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);

        consumer.subscribe(Arrays.asList("enbook", "t2"));

        try {
                while (true) {
                  ConsumerRecords<String, String> records = consumer.poll(Long.MAX_VALUE);
                  for (ConsumerRecord<String, String> record : records)
                    System.out.println("c1消费:"+record.offset() + ":" + record.value());
                 }
                } catch (Exception e) {
                } finally {
                 consumer.close();
                }
}

@Test
public void consumer_2(){
        Properties props = new Properties();
        props.put("bootstrap.servers", "192.168.234.11:9092");
```

```
        props.put("group.id", "consumer-tutorial");
        props.put("key.deserializer", StringDeserializer.class.getName());
        props.put("value.deserializer", StringDeserializer.class.getName());
        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);

        consumer.subscribe(Arrays.asList("enbook", "t2"));

        try {
            while (true) {
              ConsumerRecords<String, String> records = consumer.poll(Long.MAX_VALUE);
              for (ConsumerRecord<String, String> record : records)
                System.out.println("c2消费:"+record.offset() + ":" + record.value());
            }
          } catch (Exception e) {
            } finally {
            consumer.close();
          }
    }
}
```

重点：Kafka的消息系统语义

2018年10月4日 11:21

概述

在一个分布式发布订阅消息系统中，组成系统的计算机总会由于各自的故障而不能工作。在Kafka中，一个单独的broker，可能会在生产者发送消息到一个topic的时候宕机，或者出现网络故障，从而导致生产者发送消息失败。根据生产者如何处理这样的失败，产生了不通的语义：

至少一次语义（At least once semantics）：

如果生产者收到了Kafka broker的确认（acknowledgement，ack），并且生产者的acks配置项设置为all（或-1），这就意味着消息已经被精确一次写入Kafka topic了。然而，如果生产者接收ack超时或者收到了错误，它就会认为消息没有写入Kafka topic而尝试重新发送消息。如果broker恰好在消息已经成功写入Kafka topic后，发送ack前，出了故障，生产者的重试机制就会导致这条消息被写入Kafka两次，从而导致同样的消息会被消费者消费不止一次。每个人都喜欢一个兴高采烈的给予者，但是这种方式会导致重复的工作和错误的结果。

至多一次语义（At most once semantics）：

如果生产者在ack超时或者返回错误的时候不重试发送消息，那么消息有可能最终并没有写入Kafka topic中，因此也就不会被消费者消费到。但是为了避免重复处理的可能性，我们接受有些消息可能被遗漏处理。

精确一次语义（Exactly once semantics）：

即使生产者重试发送消息，也只会让消息被发送给消费者一次。精确一次语义是最令人满意的保证，但也是最难理解的。因为它需要消息系统本身和生产消息的应用程序还有消费消息的应用程序一起合作。比如，在成功消费一条消息后，你又把消费的offset重置到之前的某个offset位置，那么你将收到从那个offset到最新的offset之间的所有消息。这解释了为什么消息系统和客户端程序必须合作来保证精确一次语义。

潜在的问题

假设有一个单进程生产者程序，发送了消息“Hello Kafka“给一个叫做“EoS“的单分区Kafka topic。然后有一个单实例的消费者程序在另一端从topic中拉取消息，然后打印。在没有故障的理想情况下，这能很好的工作，“Hello Kafka“只被写入到EoS topic一次。消费者拉取消息，处理消息，提交偏移量来说明它完成了处理。然后，即使消费者程序出故障重启也不会再收到“Hello Kafka“这条消息了。

然而，我们知道，我们不能总认为一切都是顺利的。在上规模的集群中，即使最不可能发生的故障场景都可能最终发生。比如：

1.broker可能故障：

Kafka是一个高可用、持久化的系统，每一条写入一个分区的消息都会被持久化并且多副本备份（假设有n个副本）。所以，Kafka可以容忍n-1哥broker故障，意味着一个分区只要至少有一个broker可用，分区就可用。Kafka的副本协议保证了只要消息被成功写入了主副本，它就会被复制到其他所有的可用副本（ISR）。

2.producer到broker的RPC调用可能失败：

Kafka的持久性依赖于生产者接收broker的ack。没有接收成功ack不代表生产请求本身失败了。broker可能在写入消息后，发送ack给生产者的时候挂了。甚至broker也可能在写入消息前就挂了。由于生产者没有办法知道错误是什么造成的，所以它就只能认为消息没写入成功，并且会重试发送。在一些情况下，这会造成同样的消息在Kafka分区日志中重复，进而造成消费端多次收到这条消息。

新版本Kafka的幂等性实现

一个幂等性的操作就是一种被执行多次造成的影响和只执行一次造成的影响一样的操作。现在生产者发送的操作是幂等的了。如果出现导致生产者重试的错误，同样的消息，仍由同样的生产者发送多次，将只被写到kafka broker的日志中一次。对于单个分区，幂等生产者不会因为生产者或broker故障而发送多条重复消

息。想要开启这个特性，获得每个分区内的精确一次语义，也就是说没有重复，没有丢失，并且有序的语义，**只需要设置producer配置中的”enable.idempotence=true”**。

这个特性是怎么实现的呢？在底层，它和TCP的工作原理有点像，每发送到Kafka的消息都将包含一个序列号，broker将使用这个序列号来删除重复的发送。和只能在瞬态内存中的连接中保证不重复的TCP不同，这个序列号被持久化到副本日志，所以，即使分区的leader挂了，其他的broker接管了leader，新leader仍可以判断重新发送的是否重复了。这种机制的开销非常低：每批消息只有几个额外的字段。你将在这篇文章的后面看到，这种特性比非幂等的生产者只增加了可忽略的性能开销。

此外，Kafka除了构建于生产者—>broker的幂等性之外，从broker->消费者的精确一次流处理现在可以通过Apache Kafka的流处理API实现了。

你仅需要设置配置：“processing.guarantee = exact_once”。 这可以保证消费者的所有处理恰好发生一次。

这就是为什么Kafka的Streams API提供的精确一次性保证是迄今为止任何流处理系统提供的最强保证。 它为流处理应用程序提供端到端的一次性保证，从Kafka读取的数据，Streams应用程序物化到Kafka的任何状态，到写回Kafka的最终输出。 仅依靠外部数据系统来实现状态支持的流处理系统对于精确一次的流处理提供了较少的保证。 即使他们使用Kafka作为流处理的源并需要从失败中恢复，他们也只能倒回他们的Kafka偏移量来重建和重新处理消息，但是不能回滚外部系统中的关联状态，导致状态不正确，更新不是幂等的。

通过代码来设置消息系统语义

Producer的至多一次：

```
@Test
public void producer() throws  ExecutionException{
   Properties props=new Properties();
   props.put("key.serializer", "org.apache.kafka.common.serialization.IntegerSerializer");
```

```
    props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
    props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,"192.168.150.137:9092,192.168.150.138:9092");
    props.put("acks", 0);

        Producer<Integer, String> kafkaProducer = new KafkaProducer<Integer, String>(props);
        for(int i=0;i<100;i++){
            ProducerRecord<Integer, String> message
             = new ProducerRecord<Integer, String>("jpbook",""+i);
            kafkaProducer.send(message);
        }

        while(true);
}
```

Producer的至少一次:

```
@Test
public void producer() throws  ExecutionException{
    Properties props=new Properties();
    props.put("key.serializer", "org.apache.kafka.common.serialization.IntegerSerializer");
    props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
    props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,"192.168.150.137:9092,192.168.150.138:9092");
    props.put("acks", all);

        Producer<Integer, String> kafkaProducer = new KafkaProducer<Integer, String>(props);
        for(int i=0;i<100;i++){
            ProducerRecord<Integer, String> message
             = new ProducerRecord<Integer, String>("jpbook",""+i);
            kafkaProducer.send(message);
        }

        while(true);
}
```

Producer的精确一次:

```
@Test
public void producer() throws  ExecutionException{
```

```
Properties props=new Properties();
props.put("key.serializer", "org.apache.kafka.common.serialization.IntegerSerializer");
props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,"192.168.150.137:9092,192.168.150.138:9092");
props.put("acks","all");
props.put("enable.idempotence","true");

Producer<Integer, String> kafkaProducer = new KafkaProducer<Integer, String>(props);
    for(int i=0;i<100;i++){
        ProducerRecord<Integer, String> message
         = new ProducerRecord<Integer, String>("jpbook",""+i);
        kafkaProducer.send(message);
    }

    while(true);

}
```

Consumer的至多一次：

kafka consumer是默认至多一次，consumer的配置是：

1. 设置enable.auto.commit 为 true.
2. 设置 auto.commit.interval.ms为一个较小的值.
3. consumer不去执行 consumer.commitSync()， 这样, Kafka 会每隔一段时间自动提交offset。

```
@Test
public void consumer_2(){
    Properties props = new Properties();
    props.put("bootstrap.servers", "192.168.150.137:9092,192.168.150.138:9092");

    props.put("group.id", "g1");
    props.put("key.deserializer", StringDeserializer.class.getName());
    props.put("value.deserializer", StringDeserializer.class.getName());
    props.put("enable.auto.commit", "true");
    props.put("auto.commit.interval.ms", "101");
    KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);

    consumer.subscribe(Arrays.asList("enbook", "t2"));

    try {
```

```
            while (true) {
              ConsumerRecords<String, String> records = consumer.poll(Long.MAX_VALUE);
              for (ConsumerRecord<String, String> record : records)
                System.out.println("g1组c2消费者,分区编号:"+record.partition()+"offset:"+record.offset() + ":" +
            record.value());
             }
            } catch (Exception e) {
            } finally {
             consumer.close();
            }
}
```

Consumer的至少一次:

设置enable.auto.commit 为 false 或者

设置enable.auto.commit为 true 并设置auto.commit.interval.ms为一个较大的值.

处理完后consumer调用 consumer.commitSync()

```
@Test
public void consumer_2(){
    Properties props = new Properties();
    props.put("bootstrap.servers", "192.168.150.137:9092,192.168.150.138:9092");
    props.put("group.id", "g1");
    props.put("key.deserializer", StringDeserializer.class.getName());
    props.put("value.deserializer", StringDeserializer.class.getName());
    props.put("enable.auto.commit", "false");
    KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);

    consumer.subscribe(Arrays.asList("enbook", "t2"));

    try {
        while (true) {
          ConsumerRecords<String, String> records = consumer.poll(Long.MAX_VALUE);
          for (ConsumerRecord<String, String> record : records)
            //--Process.....
            //--处理完成后，用户自己手动提交offset
            consumer.commitAsync();
```

```
            }
        } catch (Exception e) {
        } finally {
            consumer.close();
        }
}
```

Consumer的精确一次：

```
@Test
public void consumer_2(){
    Properties props = new Properties();
    props.put("bootstrap.servers", "192.168.150.137:9092,192.168.150.138:9092");
    props.put("group.id", "g1");
    props.put("key.deserializer", StringDeserializer.class.getName());
    props.put("value.deserializer", StringDeserializer.class.getName());
    props.put("enable.auto.commit", "false");
    props.put("processing.guarantee","exact_once");
    KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);

    consumer.subscribe(Arrays.asList("enbook", "t2"));

    try {
        while (true) {
            ConsumerRecords<String, String> records = consumer.poll(Long.MAX_VALUE);
            for (ConsumerRecord<String, String> record : records)
                //--Process.....
                //--处理完成后，用户自己手动提交offset
                consumer.commitAsync();

        }
    } catch (Exception e) {
    } finally {
        consumer.close();
    }
}
```

Kafka在Zookeeper的路径

2018年9月30日 14:46

```
rmr /cluster
rmr /brokers
rmr /admin
rmr /isr_change_notification
rmr /log_dir_event_notificatio
rmr /controller_epoch
rmr /consumers
rmr /latest_producer_id_block
rmr /config
```

Kafka配置手册

2018年10月1日 12:44

Property	Default	Description
broker.id		每个broker都可以用一个唯一的非负整数id进行标识；这个id可以作为broker的“名字”，并且它的存在使得broker无须混淆consumers就可以迁移到不同的host/port上。你可以选择任意你喜欢的数字作为id，只要id是唯一的即可。
log.dirs	/tmp/kafka-logs	kafka存放数据的路径。这个路径并不是唯一的，可以是多个，路径之间只需要使用逗号分隔即可；每当创建新partition时，都会选择在包含最少partitions的路径下进行。
zookeeper.connect	null	ZooKeeper连接字符串的格式为：hostname:port，此处hostname和port分别是ZooKeeper集群中某个节点的host和port；为了当某个host宕掉之后你能通过其他ZooKeeper节点进行连接，你可以按照一下方式制定多个hosts： hostname1:port1, hostname2:port2, hostname3:port3. ZooKeeper 允许你增加一个“chroot”路径，将集群中所有kafka数据存放在特定的路径下。当多个Kafka集群或者其他应用使用相同ZooKeeper集群时，可以使用这个方式设置数据存放路径。这种方式的实现可以通过这样设置连接字符串格式，如下所示： hostname1：port1，hostname2：port2，hostname3：port3/chroot/path 这样设置就将所有kafka集群数据存放在/chroot/path路径下。注意，在你启动broker之前，你必须创建这个路径，并且consumers必须使用相同的连接格式。
num.partitions	1	如果创建topic时没有给出划分partitions个数，这个数字将是topic下partitions数目的默认数值。 **主题的分区数直接决定了消费的并行度**，所以在实际生产中，分区数一般都在几十个或几百个。 此外注意：提供分区的副本数，并不能提高并发度，因为无论是生成者还是消费者，都是和副本的Leader交互。
log.segment.bytes	1024*1024*1024	topic partition的日志存放在某个目录下诸多文件中，这些文件将partition的日志切分成一段一段的；这个属性就是每个文件的最大尺寸；当尺寸达到这个数值时，就会创建新文件。此设置可以由每个topic基础设置时进行覆盖。
log.roll.hours	24 * 7	即使文件没有到达log.segment.bytes，只要文件创建时间到达此属性，就会创建新文件。这个设置也可以有topic层面的设置进行覆盖；
log.index.size.max.bytes	10*1024*1024	每个log segment的最大尺寸。注意，如果log尺寸达到这个数值，即使尺寸没有超过log.segment.bytes限制，也需要产生新的log segment。
replica.lag.time.max.ms	10000	如果一个follower在这个时间内没有发送fetch请求，leader将从ISR中移除这个follower

replica.fetch.max.bytes	1024*1024	备份时每次fetch的最大值
unclean.leader.election.enable	true	指明了是否能够使不在ISR中replicas设置用来作为leader
delete.topic.enable	false	能够删除topic

boostrap.servers		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	1	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。
batch.size	16384（字节） 即：16kb	producer将试图批处理消息记录，以减少请求次数。这项配置控制默认的批量处理消息字节数。 此参数控制的是生成者的批处理大小，此参数越大，会使生成的吞吐量越大，同时也会占用过大的内存空间。 注意：调节测试是1024整数倍
linger.ms	0	Producer默认会把两次发送时间间隔内收集到的所有Requests进行一次聚合然后再发送，以此提高吞吐量，而linger.ms则更进一步，这个参数为每次发送增加一些delay，以此来聚合更多的Message。官网解释翻译：producer会将request传输之间到达的所有records聚合到一个批请求。通常这个值发生在欠负载情况下，record到达速度快于发送。但是在某些场景下，client即使在正常负载下也期望减少请求数量。这个设置就是如此，通过人工添加少量时延，而不是立马发送一个record，producer会等待所给的时延，以让其他records发送出去，这样就会被聚合在一起。这个类似于TCP的Nagle算法。该设置给了batch的时延上限：当我们获得一个partition的batch.size大小的records，就会立即发送出去，而不管该设置；但是如果对于这个partition没有累积到足够的record，会linger指定的时间等待更多的records出现。该设置的默认值为0(无时延)

2019年7月16日 17:22

Kafka Connect 是 Apache Kafka 的一部分，为其他数据存储和 Kafka 提供流式集成。对于数据工程师来说，他们只需要配置一下 JSON 文件就可以了。Kafka 提供了一些可用于常见数据存储的连接器，如 JDBC、Elasticsearch、IBM MQ、S3 和 BigQuery，等等。

对于开发人员来说，Kafka Connect 提供了丰富的 API，如果有必要还可以开发其他连接器。除此之外，它还提供了用于配置和管理连接器的 REST API。

Kafka Connect 是一种模块化组件，提供了一种非常强大的集成方法。一些关键组件包括：

连接器——定义如何与数据存储集成的 JAR 文件；
转换器——处理数据的序列化和反序列化；
变换——可选的运行时消息操作。
人们对 Kafka Connect 最常见的误解与数据的序列化有关。Kafka Connect 使用转换器处理数据序列化。接下来让我们看看它们是如何工作的，并说明如何解决一些常见问题。

Kafka 消息都是字节
Kafka 消息被保存在主题中，每条消息就是一个键值对。当它们存储在 Kafka 中时，键和值都只是字节。Kafka 因此可以适用于各种场景，但这也意味着开发人员需要决定如何序列化数据。

在配置 Kafka Connect 时，序列化格式是最关键的配置选项之一。你需要确保从主题读取数据时使用的序列化格式与写入主题的序列化格式相同，否则就会出现混乱和错误！

序列化格式有很多种，常见的包括：

JSON；
Avro；
Protobuf；
字符串分隔（如 CSV）。
选择序列化格式
选择序列化格式的一些指导原则：

schema。很多时候，你的数据都有对应的 schema。你可能不喜欢，但作为开发人员，你有责任保留和传播 schema。schema 为服务之间提供了一种契约。某些消息格式（例如 Avro 和 Protobuf）具有强大的 schema 支持，而其他消息格式支持较少（JSON）或根本没有（CVS）。
生态系统兼容性。Avro 是 Confluent 平台的一等公民，拥有来自 Confluent Schema

Registry、Kafka Connect、KSQL 的原生支持。另一方面，Protobuf 依赖社区为部分功能提供支持。
消息大小。JSON 是纯文本的，并且依赖了 Kafka 本身的压缩机制，Avro 和 Protobuf 都是二进制格式，序列化的消息体积更小。
语言支持。Avro 在 Java 领域得到了强大的支持，但如果你的公司不是基于 Java 的，那么可能会觉得它不太好用。
如果目标系统使用 JSON，Kafka 主题也必须使用 JSON 吗?
完全不需要这样。从数据源读取数据或将数据写入外部数据存储的格式不需要与 Kafka 消息的序列化格式一样。

Kafka Connect 中的连接器负责从源数据存储（例如数据库）获取数据，并以数据内部表示将数据传给转换器。然后，Kafka Connect 的转换器将这些源数据对象序列化到主题上。

在使用 Kafka Connect 作为接收器时刚好相反——转换器将来自主题的数据反序列化为内部表示，传给连接器，以便能够使用特定于目标的适当方法将数据写入目标数据存储。

也就是说，主题数据可以是 Avro 格式，当你将数据写入 HDFS 时，指定接收器的连接器使用 HDFS 支持的格式即可。

配置转换器
Kafka Connect 默认使用了 worker 级别的转换器配置，连接器可以对其进行覆盖。由于在整个管道中使用相同的序列化格式通常会更好，所以一般只需要在 worker 级别设置转换器，而不需要在连接器中指定。但你可能需要从别人的主题拉取数据，而他们使了用不同的序列化格式——对于这种情况，你需要在连接器配置中设置转换器。即使你在连接器的配置中进行了覆盖，它仍然是执行实际任务的转换器。

好的连接器一般不会序列化或反序列化存储在 Kafka 中的消息，它会让转换器完成这项工作。

请记住，Kafka 消息是键值对字节，你需要使用 key.converter 和 value.converter 为键和值指定转换器。在某些情况下，你可以为键和值使用不同的转换器。

这是使用 String 转换器的一个示例。

复制代码

```
"key.converter": "org.apache.kafka.connect.storage.StringConverter",
```

有些转换器有一些额外的配置。对于 Avro，你需要指定 Schema Registry。对于 JSON，你需要指定是否希望 Kafka Connect 将 schema 嵌入到 JSON 消息中。在指定特定于转换器的配置时，

请始终使用 key.converter. 或 value.converter. 前缀。例如，要将 Avro 用于消息载荷，你需要指定以下内容：

复制代码

"value.converter": "io.confluent.connect.avro.AvroConverter",

"value.converter.schema.registry.url": "http://schema-registry:8081",

常见的转换器包括：

Avro——来自 Confluent 的开源项目
复制代码

io.confluent.connect.avro.AvroConverter

String——Apache Kafka 的一部分
复制代码

org.apache.kafka.connect.storage.StringConverter

JSON——Apache Kafka 的一部分
复制代码

org.apache.kafka.connect.json.JsonConverter

ByteArray——Apache Kafka 的一部分
复制代码

org.apache.kafka.connect.converters.ByteArrayConverter

Protobuf——来自社区的开源项目
复制代码

com.blueapron.connect.protobuf.ProtobufConverter

JSON 和 schema

虽然 JSON 默认不支持嵌入 schema，但 Kafka Connect 提供了一种可以将 schema 嵌入到消息中的特定 JSON 格式。由于 schema 被包含在消息中，因此生成的消息大小可能会变大。

如果你正在设置 Kafka Connect 源，并希望 Kafka Connect 在写入 Kafka 消息包含 schema，可以这样：

复制代码

```
value.converter=org.apache.kafka.connect.json.JsonConverter

value.converter.schemas.enable=true
```

生成的 Kafka 消息看起来像下面这样，其中包含 schema 和 payload 节点元素：

复制代码

```
{
  "schema": {
    "type": "struct",
    "fields": [
      {
        "type": "int64",
        "optional": false,
        "field": "registertime"
      },
```

```
    {
      "type": "string",
      "optional": false,
      "field": "userid"
    },
    {
      "type": "string",
      "optional": false,
      "field": "regionid"
    },
    {
      "type": "string",
      "optional": false,
      "field": "gender"
    }
  ],
  "optional": false,
  "name": "ksql.users"
},
```

```
 "payload": {
  "registertime": 1493819497170,
  "userid": "User_1",
  "regionid": "Region_5",
  "gender": "MALE"
 }
}
```

请注意消息的大小，消息由 playload 和 schema 组成。每条消息中都会重复这些数据，这也就是为什么说 Avro 这样的格式会更好，因为它的 schema 是单独存储的，消息中只包含载荷（并进行了压缩）。

如果你正在使用 Kafka Connect 消费 Kafka 主题中的 JSON 数据，那么就需要知道数据是否包含了 schema。如果包含了，并且它的格式与上述的格式相同，那么你可以这样设置：

复制代码

```
value.converter=org.apache.kafka.connect.json.JsonConverter
value.converter.schemas.enable=true
```

不过，如果你正在消费的 JSON 数据如果没有 schema 加 payload 这样的结构，例如：

复制代码

```
{
 "registertime": 1489869013625,
 "userid": "User_1",
```

```
 "regionid": "Region_2",

 "gender": "OTHER"

}
```

那么你必须通过设置 schemas.enable = false 告诉 Kafka Connect 不要查找 schema：

复制代码

```
value.converter=org.apache.kafka.connect.json.JsonConverter

value.converter.schemas.enable=false
```

和之前一样，转换器配置选项（这里是 schemas.enable）需要使用前缀 key.converter 或 value.converter。

常见错误

如果你错误地配置了转换器，将会遇到以下的一些常见错误。这些消息将显示在你为 Kafka Connect 配置的接收器中，因为你试图在接收器中反序列化 Kafka 消息。这些错误会导致连接器失败，主要错误消息如下所示：

复制代码

```
ERROR WorkerSinkTask{id=sink-file-users-json-noschema-01-0} Task threw an uncaught and
unrecoverable exception (org.apache.kafka.connect.runtime.WorkerTask)

org.apache.kafka.connect.errors.ConnectException: Tolerance exceeded in error handler

  at org.apache.kafka.connect.runtime.errors.RetryWithToleranceOperator.
execAndHandleError(RetryWithToleranceOperator.java:178)

  at org.apache.kafka.connect.runtime.errors.RetryWithToleranceOperator.execute
(RetryWithToleranceOperator.java:104)
```

在错误消息的后面，你将看到堆栈信息，描述了发生错误的原因。请注意，对于连接器中的任何致命错误，都会抛出上述异常，因此你可能会看到与序列化无关的错误。要快速查看错误配置可能会导致的错误，请参考下表：

问题：使用 JsonConverter 读取非 JSON 数据
如果你的源主题上有非 JSON 数据，但尝试使用 JsonConverter 读取它，你将看到：

复制代码

```
org.apache.kafka.connect.errors.DataException: Converting byte[] to Kafka Connect data failed due to serialization error:

…

org.apache.kafka.common.errors.SerializationException: java.io.CharConversionException: Invalid UTF-32 character 0x1cfa7e2 (above 0x0010ffff) at char #1, byte #7)
```

这有可能是因为源主题使用了 Avro 或其他格式。

解决方案：如果数据是 Avro 格式的，那么将 Kafka Connect 接收器的配置改为：

复制代码

```
"value.converter": "io.confluent.connect.avro.AvroConverter",

"value.converter.schema.registry.url": "http://schema-registry:8081",
```

或者，如果主题数据是通过 Kafka Connect 填充的，那么你也可以这么做，让上游源也发送 JSON 数据：

复制代码

```
"value.converter": "org.apache.kafka.connect.json.JsonConverter",

"value.converter.schemas.enable": "false",
```

问题：使用 AvroConverter 读取非 Avro 数据
这可能是我在 Confluent Community 邮件组和 Slack 组等地方经常看到的错误。当你尝试使用 Avro 转换器从非 Avro 主题读取数据时，就会发生这种情况。这包括使用 Avro 序列化器而不是 Confluent Schema Registry 的 Avro 序列化器（它有自己的格式）写入的数据。

复制代码

```
org.apache.kafka.connect.errors.DataException: my-topic-name

at io.confluent.connect.avro.AvroConverter.toConnectData(AvroConverter.java:97)

...

org.apache.kafka.common.errors.SerializationException: Error deserializing Avro message for id -1

org.apache.kafka.common.errors.SerializationException: Unknown magic byte!
```

解决方案：检查源主题的序列化格式，修改 Kafka Connect 接收器连接器，让它使用正确的转换器，或将上游格式切换为 Avro。如果上游主题是通过 Kafka Connect 填充的，则可以按如下方式配置源连接器的转换器：

复制代码

```
"value.converter": "io.confluent.connect.avro.AvroConverter",

"value.converter.schema.registry.url": "http://schema-registry:8081",
```

问题：没有使用预期的 schema/payload 结构读取 JSON 消息
如前所述，Kafka Connect 支持包含载荷和 schema 的 JSON 消息。如果你尝试读取不包含这种结构的 JSON 数据，你将收到这个错误：

复制代码

```
org.apache.kafka.connect.errors.DataException: JsonConverter with schemas.enable requires "schema" and "payload" fields and may not contain additional fields. If you are trying to deserialize plain JSON data, set schemas.enable=false in your converter configuration.
```

需要说明的是，当 schemas.enable=true 时，唯一有效的 JSON 结构需要包含 schema 和 payload 这两个顶级元素（如上所示）。

如果你只有简单的 JSON 数据，则应将连接器的配置改为：

复制代码

"value.converter": "org.apache.kafka.connect.json.JsonConverter",

"value.converter.schemas.enable": "false",

如果要在数据中包含 schema，可以使用 Avro（推荐），也可以修改上游的 Kafka Connect 配置，让它在消息中包含 schema：

复制代码

"value.converter": "org.apache.kafka.connect.json.JsonConverter",

"value.converter.schemas.enable": "true",

故障排除技巧
查看 Kafka Connect 日志
要在 Kafka Connect 中查找错误日志，你需要找到 Kafka Connect 工作程序的输出。这个位置取决于你是如何启动 Kafka Connect 的。有几种方法可用于安装 Kafka Connect，包括 Docker、Confluent CLI、systemd 和手动下载压缩包。你可以这样查找日志的位置：

Docker：docker logs container_name；
Confluent CLI：confluent log connect；
systemd：日志文件在 /var/log/confluent/kafka-connect；
其他：默认情况下，Kafka Connect 将其输出发送到 stdout，因此你可以在启动 Kafka Connect 的终端中找到它们。
查看 Kafka Connect 配置文件
Docker——设置环境变量，例如在 Docker Compose 中：
复制代码

CONNECT_KEY_CONVERTER: io.confluent.connect.avro.AvroConverter

CONNECT_KEY_CONVERTER_SCHEMA_REGISTRY_URL: 'http://schema-registry:8081'

CONNECT_VALUE_CONVERTER: io.confluent.connect.avro.AvroConverter

CONNECT_VALUE_CONVERTER_SCHEMA_REGISTRY_URL: 'http://schema-registry:8081'

Confluent CLI——使用配置文件 etc/schema-registry/connect-avro-distributed.properties；
systemd（deb/rpm）——使用配置文件 /etc/kafka/connect-distributed.properties；
其他——在启动 Kafka Connect 时指定工作程序的属性文件，例如：
复制代码

```
$ cd confluent-5.0.0

$ ./bin/connect-distributed ./etc/kafka/connect-distributed.properties
```

检查 Kafka 主题
假设我们遇到了上述当中的一个错误，并且想要解决 Kafka Connect 接收器无法从主题读取数据的问题。

我们需要检查正在被读取的数据，并确保它使用了正确的序列化格式。另外，所有消息都必须使用这种格式，所以不要假设你现在正在以正确的格式向主题发送消息就不会出问题。Kafka Connect 和其他消费者也会从主题上读取已有的消息。

下面，我将使用命令行进行故障排除，当然也可以使用其他的一些工具：

Confluent Control Center 提供了可视化检查主题内容的功能；
KSQL 的 PRINT 命令将主题的内容打印到控制台；
Confluent CLI 工具提供了 consume 命令，可用于读取字符串和 Avro 数据。
如果你的数据是字符串或 JSON 格式
你可以使用控制台工具，包括 kafkacat 和 kafka-console-consumer。我个人的偏好是使用 kafkacat：

复制代码

```
$ kafkacat -b localhost:9092 -t users-json-noschema -C -c1

{"registertime":1493356576434,"userid":"User_8","regionid":"Region_2","gender":"MALE"}
```

你也可以使用 jq 验证和格式化 JSON：

复制代码

```
$ kafkacat -b localhost:9092 -t users-json-noschema -C -c1|jq '.'
```

```
{
 "registertime": 1493356576434,
 "userid": "User_8",
 "regionid": "Region_2",
 "gender": "MALE"
}
```

如果你得到一些“奇怪的”字符，你查看的很可能是二进制数据，这些数据是通过 Avro 或 Protobuf 写入的：

复制代码

```
$ kafkacat -b localhost:9092 -t users-avro -C -c1
ص♦♦♦User_9Region_MALE
```

如果你的数据是 Avro 格式
你应该使用专为读取和反序列化 Avro 数据而设计的控制台工具。我使用的是 kafka-avro-console-consumer。确保指定了正确的 Schema Registry URL：

复制代码

```
$ kafka-avro-console-consumer --bootstrap-server localhost:9092 \
                --property schema.registry.url=http://localhost:8081 \
                --topic users-avro \
                --from-beginning --max-messages 1
{"registertime":1505213905022,"userid":"User_5","regionid":"Region_4","gender":"FEMALE"}
```

和之前一样，如果要格式化，可以使用 jq：

复制代码

```
$ kafka-avro-console-consumer --bootstrap-server localhost:9092 \

                --property schema.registry.url=http://localhost:8081 \

                --topic users-avro \

                --from-beginning --max-messages 1 | \

                jq '.'

{

 "registertime": 1505213905022,

 "userid": "User_5",

 "regionid": "Region_4",

 "gender": "FEMALE"

}
```

内部转换器

在分布式模式下运行时，Kafka Connect 使用 Kafka 来存储有关其操作的元数据，包括连接器配置、偏移量等。

可以通过 internal.key.converter/internal.value.converter 让这些 Kafka 使用不同的转换器。不过这些设置只在内部使用，实际上从 Apache Kafka 2.0 开始就已被弃用。你不应该更改这些配置，从 Apache Kafka 2.0 版开始，如果你这么做了将会收到警告。

将 schema 应用于没有 schema 的消息

很多时候，Kafka Connect 会从已经存在 schema 的地方引入数据，并使用合适的序列化格式（例如 Avro）来保留这些 schema。然后，这些数据的所有下游用户都可以使用这些 schema。但如果没有提供显式的 schema 该怎么办？

或许你正在使用 FileSourceConnector 从普通文件中读取数据（不建议用于生产环境中，但可用于 PoC），或者正在使用 REST 连接器从 REST 端点提取数据。由于它们都没有提供 schema，因此你需要声明它。

有时候你只想传递你从源读取的字节，并将它们保存在一个主题上。但大多数情况下，你需要 schema 来使用这些数据。在摄取时应用一次 schema，而不是将问题推到每个消费者，这才是一种更好的处理方式。

你可以编写自己的 Kafka Streams 应用程序，将 schema 应用于 Kafka 主题中的数据上，当然你也可以使用 KSQL。下面让我们来看一下将 schema 应用于某些 CSV 数据的简单示例。

假设我们有一个 Kafka 主题 testdata-csv，保存着一些 CSV 数据，看起来像这样：

复制代码

```
$ kafkacat -b localhost:9092 -t testdata-csv -C

1,Rick Astley,Never Gonna Give You Up

2,Johnny Cash,Ring of Fire
```

我们可以猜测它有三个字段，可能是：

```
ID
Artist
Song
```

如果我们将数据保留在这样的主题中，那么任何想要使用这些数据的应用程序——无论是 Kafka Connect 接收器还是自定义的 Kafka 应用程序——每次都需要都猜测它们的 schema 是什么。或者，每个消费应用程序的开发人员都需要向提供数据的团队确认 schema 是否发生变更。正如 Kafka 可以解耦系统一样，这种 schema 依赖让团队之间也有了硬性耦合，这并不是一件好事。

因此，我们要做的是使用 KSQL 将 schema 应用于数据上，并使用一个新的派生主题来保存 schema。这样你就可以通过 KSQL 检查主题数据：

复制代码

```
ksql> PRINT 'testdata-csv' FROM BEGINNING;
```

Format:STRING

11/6/18 2:41:23 PM UTC , NULL , 1,Rick Astley,Never Gonna Give You Up

11/6/18 2:41:23 PM UTC , NULL , 2,Johnny Cash,Ring of Fire

前两个字段（11/6/18 2:41:23 PM UTC 和 NULL）分别是 Kafka 消息的时间戳和键。其余字段来自 CSV 文件。现在让我们用 KSQL 注册这个主题并声明 schema：

复制代码

```
ksql> CREATE STREAM TESTDATA_CSV (ID INT, ARTIST VARCHAR, SONG VARCHAR) \

WITH (KAFKA_TOPIC='testdata-csv', VALUE_FORMAT='DELIMITED');

Message

----------------

Stream created

----------------
```

可以看到，KSQL 现在有一个数据流 schema：

复制代码

```
ksql> DESCRIBE TESTDATA_CSV;

Name                 : TESTDATA_CSV

 Field   | Type

-------------------------------------

 ROWTIME | BIGINT (system)
```

```
 ROWKEY  | VARCHAR(STRING) (system)

 ID      | INTEGER

 ARTIST  | VARCHAR(STRING)

 SONG    | VARCHAR(STRING)

-------------------------------------
```

For runtime statistics and query details run: DESCRIBE EXTENDED <Stream,Table>;

可以通过查询 KSQL 流来检查数据是否符合预期。请注意，这个时候我们只是作为现有 Kafka 主题的消费者——并没有更改或复制任何数据。

复制代码

```
ksql> SET 'auto.offset.reset' = 'earliest';

Successfully changed local property 'auto.offset.reset' from 'null' to 'earliest'

ksql> SELECT ID, ARTIST, SONG FROM TESTDATA_CSV;

1 | Rick Astley | Never Gonna Give You Up

2 | Johnny Cash | Ring of Fire
```

最后，创建一个新的 Kafka 主题，使用带有 schema 的数据进行填充。KSQL 查询是持续的，因此除了将现有的数据从源主题发送到目标主题之外，KSQL 还将向目标主题发送未来将生成的数据。

复制代码

```
ksql> CREATE STREAM TESTDATA WITH (VALUE_FORMAT='AVRO') AS SELECT * FROM
TESTDATA_CSV;
```

Message

Stream created and running

使用 Avro 控制台消费者验证数据：

复制代码

```
$ kafka-avro-console-consumer --bootstrap-server localhost:9092 \
                              --property schema.registry.url=http://localhost:8081 \
                              --topic TESTDATA \
                              --from-beginning | \
                              jq '.'
{
  "ID": {
    "int": 1
},
  "ARTIST": {
    "string": "Rick Astley"
},
  "SONG": {
```

```
    "string": "Never Gonna Give You Up"
  }
}
[…]
```

你甚至可以在 Schema Registry 中查看已注册的 schema：

复制代码

```
$ curl -s http://localhost:8081/subjects/TESTDATA-value/versions/latest|jq '.schema|fromjson'
{
  "type": "record",
  "name": "KsqlDataSourceSchema",
  "namespace": "io.confluent.ksql.avro_schemas",
  "fields": [
    {
      "name": "ID",
      "type": [
        "null",
        "int"
      ],
```

```
    "default": null
  },
  {
    "name": "ARTIST",
    "type": [
      "null",
      "string"
    ],
    "default": null
  },
  {
    "name": "SONG",
    "type": [
      "null",
      "string"
    ],
    "default": null
  }
]
```

}

写入原始主题（testdata-csv）的任何新消息都由 KSQL 自动处理，并以 Avro 格式写入新的 TESTDATA 主题。现在，任何想要使用这些数据的应用程序或团队都可以使用 TESTDATA 主题。你还可以更改主题的分区数、分区键和复制系数。

原文：

https://blog.csdn.net/D55dffdh/article/details/84929263

2019年7月23日 8:42

sparkstreaming + kafka如何保证数据不丢失、不重复

2016年12月03日 19:03:36 hadooper-enjoyment 阅读数 3522

spark-streaming作为一个24*7不间断运行的程序来设计，但是程序都会crash，如果crash了，如何保证数据不丢失，不重复。

Input DStreams and Receivers

spark streaming提供了两种streaming input source：

1. basic source: Source directly avaliable in the StreamingContext API. Examples: file,socket connnection
2. advanced source: Source like kafka/kinesis, etc. are avaliable through extra utility classes.

本文只讨论高级数据源，因为针对流计算场景，基本数据源不适用。

高级数据源，这里以kafka为例，kafka作为输入源，有两种方式：

1. Receiver-based 方式

2. Direct 方式

两种方式的对比见博客：

保证数据不丢失（at-least）

spark RDD内部机制可以保证数据at-least语义。

Receiver方式

开启WAL（预写日志），将从kafka中接受到的数据写入到日志文件中，所有数据从失败中可恢复。

Direct方式

依靠checkpoint机制来保证。

保证数据不重复（exactly-once）

要保证数据不重复，即Exactly once语义。

- 幂等操作：重复执行不会产生问题，不需要做额外的工作即可保证数据不重复。

- 业务代码添加事务操作

```
dstream.foreachRDD {(rdd, time) =
  rdd.foreachPartition { partitionIterator =>
    val partitionId = TaskContext.get.partitionId()
    val uniqueId = generateUniqueId(time.milliseconds,partitionId)
    //use this uniqueId to transationally commit the data in partitionIterator
 }
7
```

就是说针对每个partition的数据，产生一个uniqueId，只有这个partition的所有数据被完全消费，则算成功，否则算失效，要回滚。下次重复执行这个uniqueId时，如果已经被执行成功，则skip掉。

来自 <https://blog.csdn.net/enjoy524/article/details/53446893>

`offsetRanges`

来自 <https://cloud.tencent.com/developer/article/1150861>

2.4.3

- Overview
- Programming Guides
- API Docs
- Deploying
- More

Spark Streaming + Kafka Integration Guide (Kafka broker version 0.10.0 or higher)

The Spark Streaming integration for Kafka 0.10 is similar in design to the 0.8 Direct Stream approach. It provides simple parallelism, 1:1 correspondence between Kafka partitions and Spark partitions, and access to offsets and metadata. However, because the newer integration uses the new Kafka consumer API instead of the simple API, there are notable differences in usage. This version of the integration is marked as experimental, so the API is potentially subject to change.

Linking

For Scala/Java applications using SBT/Maven project definitions, link your streaming application with the following artifact (see Linking section in the main programming guide for further information).

```
groupId = org.apache.spark
artifactId = spark-streaming-kafka-0-10_2.12
version = 2.4.3
```

Do not manually add dependencies on `org.apache.kafka` artifacts (e.g. `kafka-clients`). The `spark-streaming-kafka-0-10` artifact has the appropriate transitive dependencies already, and different versions may be incompatible in hard to diagnose ways.

Creating a Direct Stream

Note that the namespace for the import includes the version, org.apache.spark.streaming.kafka010

- Scala
- Java

```
import org.apache.kafka.clients.consumer.ConsumerRecord
import org.apache.kafka.common.serialization.StringDeserializer
import org.apache.spark.streaming.kafka010._
import org.apache.spark.streaming.kafka010.LocationStrategies.PreferConsistent
import org.apache.spark.streaming.kafka010.ConsumerStrategies.Subscribe
val kafkaParams = Map[String, Object](
  "bootstrap.servers" -> "localhost:9092,anotherhost:9092",
  "key.deserializer" -> classOf[StringDeserializer],
  "value.deserializer" -> classOf[StringDeserializer],
  "group.id" -> "use_a_separate_group_id_for_each_stream",
  "auto.offset.reset" -> "latest",
  "enable.auto.commit" -> (false: java.lang.Boolean)
)

val topics = Array("topicA", "topicB")
val stream = KafkaUtils.createDirectStream[String, String](
  streamingContext,
  PreferConsistent,
  Subscribe[String, String](topics, kafkaParams)
)

stream.map(record => (record.key, record.value))
```

Each item in the stream is a ConsumerRecord

For possible kafkaParams, see Kafka consumer config docs. If your Spark batch duration is larger than the default Kafka heartbeat session timeout (30 seconds), increase heartbeat.interval.ms and session.timeout.ms appropriately. For batches larger than 5 minutes, this will require changing group.max.session.timeout.ms on the broker. Note that the example sets enable.auto.commit to false, for discussion see Storing Offsets below.

LocationStrategies

The new Kafka consumer API will pre-fetch messages into buffers. Therefore it is important for performance reasons that the Spark integration keep cached consumers on executors (rather than recreating them for each batch), and prefer to schedule partitions on the host locations that have the appropriate consumers.

In most cases, you should use `LocationStrategies.PreferConsistent` as shown above. This will distribute partitions evenly across available executors. If your executors are on the same hosts as your Kafka brokers, use `PreferBrokers`, which will prefer to schedule partitions on the Kafka leader for that partition. Finally, if you have a significant skew in load among partitions, use `PreferFixed`. This allows you to specify an explicit mapping of partitions to hosts (any unspecified partitions will use a consistent location).

The cache for consumers has a default maximum size of 64. If you expect to be handling more than (64 * number of executors) Kafka partitions, you can change this setting via `spark.streaming.kafka.consumer.cache.maxCapacity`.

If you would like to disable the caching for Kafka consumers, you can set `spark.streaming.kafka.consumer.cache.enabled` to `false`.

The cache is keyed by topicpartition and `group.id`, so use a **separate** `group.id` for each call to `createDirectStream`.

ConsumerStrategies

The new Kafka consumer API has a number of different ways to specify topics, some of which require considerable post-object-instantiation setup. `ConsumerStrategies` provides an abstraction that allows Spark to obtain properly configured consumers even after restart from checkpoint.

`ConsumerStrategies.Subscribe`, as shown above, allows you to subscribe to a fixed collection of topics. `SubscribePattern` allows you to use a regex to specify topics of interest. Note that unlike the 0.8 integration, using `Subscribe` or `SubscribePattern` should respond to adding partitions during a running stream. Finally, `Assign` allows you to specify a fixed collection of partitions. All three strategies have overloaded constructors that allow you to specify the starting offset for a particular partition.

If you have specific consumer setup needs that are not met by the options above, `ConsumerStrategy` is a public class that you can extend.

Creating an RDD

If you have a use case that is better suited to batch processing, you can create an RDD for a defined range of offsets.

- Scala
- Java

```
// Import dependencies and create kafka params as in Create Direct Stream above
val offsetRanges = Array(
  // topic, partition, inclusive starting offset, exclusive ending offset
  OffsetRange("test", 0, 0, 100),
  OffsetRange("test", 1, 0, 100)
)

val rdd = KafkaUtils.createRDD[String, String](sparkContext, kafkaParams, offsetRanges,
PreferConsistent)
```

Note that you cannot use `PreferBrokers`, because without the stream there is not a driver-side consumer to automatically look up broker metadata for you. Use `PreferFixed` with your own metadata lookups if necessary.

Obtaining Offsets

- Scala
- Java

```
stream.foreachRDD { rdd =>
  val offsetRanges = rdd.asInstanceOf[HasOffsetRanges].offsetRanges
  rdd.foreachPartition { iter =>
    val o: OffsetRange = offsetRanges(TaskContext.get.partitionId)
    println(s"${o.topic} ${o.partition} ${o.fromOffset} ${o.untilOffset}")
  }
}
```

Note that the typecast to `HasOffsetRanges` will only succeed if it is done in the first method called on the result of `createDirectStream`, not later down a chain of methods. Be aware that the one-to-one mapping between RDD partition and Kafka partition does not remain after any methods that shuffle or repartition, e.g. reduceByKey() or window().

Storing Offsets

Kafka delivery semantics in the case of failure depend on how and when offsets are

stored. Spark output operations are at-least-once. So if you want the equivalent of exactly-once semantics, you must either store offsets after an idempotent output, or store offsets in an atomic transaction alongside output. With this integration, you have 3 options, in order of increasing reliability (and code complexity), for how to store offsets.

Checkpoints

If you enable Spark checkpointing, offsets will be stored in the checkpoint. This is easy to enable, but there are drawbacks. Your output operation must be idempotent, since you will get repeated outputs; transactions are not an option. Furthermore, you cannot recover from a checkpoint if your application code has changed. For planned upgrades, you can mitigate this by running the new code at the same time as the old code (since outputs need to be idempotent anyway, they should not clash). But for unplanned failures that require code changes, you will lose data unless you have another way to identify known good starting offsets.

Kafka itself

Kafka has an offset commit API that stores offsets in a special Kafka topic. By default, the new consumer will periodically auto-commit offsets. This is almost certainly not what you want, because messages successfully polled by the consumer may not yet have resulted in a Spark output operation, resulting in undefined semantics. This is why the stream example above sets "enable.auto.commit" to false. However, you can commit offsets to Kafka after you know your output has been stored, using the `commitAsync` API. The benefit as compared to checkpoints is that Kafka is a durable store regardless of changes to your application code. However, Kafka is not transactional, so your outputs must still be idempotent.

- Scala
- Java

```
stream.foreachRDD { rdd =>
  val offsetRanges = rdd.asInstanceOf[HasOffsetRanges].offsetRanges
// some time later, after outputs have completed
  stream.asInstanceOf[CanCommitOffsets].commitAsync(offsetRanges)
}
```

As with HasOffsetRanges, the cast to CanCommitOffsets will only succeed if called on the result of createDirectStream, not after transformations. The commitAsync call is threadsafe, but must occur after outputs if you want meaningful semantics.

Your own data store

For data stores that support transactions, saving offsets in the same transaction as the results can keep the two in sync, even in failure situations. If you' re careful about detecting repeated or skipped offset ranges, rolling back the transaction prevents duplicated or lost messages from affecting results. This gives the equivalent of exactly-once semantics. It is also possible to use this tactic even for outputs that result from aggregations, which are typically hard to make idempotent.

- Scala
- Java

```
// The details depend on your data store, but the general idea looks like this
// begin from the the offsets committed to the database
val fromOffsets = selectOffsetsFromYourDatabase.map { resultSet =>
  new TopicPartition(resultSet.string("topic"), resultSet.int("partition")) -> resultSet.long("offset")
}.toMap

val stream = KafkaUtils.createDirectStream[String, String](
```

```
  streamingContext,
  PreferConsistent,
  Assign[String, String](fromOffsets.keys.toList, kafkaParams, fromOffsets)
)

stream.foreachRDD { rdd =>
  val offsetRanges = rdd.asInstanceOf[HasOffsetRanges].offsetRanges

val results = yourCalculation(rdd)

// begin your transaction

// update results
  // update offsets where the end of existing offsets matches the beginning of this batch of offsets
  // assert that offsets were updated correctly

// end your transaction
}
```

SSL / TLS

The new Kafka consumer supports SSL. To enable it, set kafkaParams appropriately before passing to createDirectStream / createRDD. Note that this only applies to communication between Spark and Kafka brokers; you are still responsible for separately securing Spark inter-node communication.

- Scala
- Java

```
val kafkaParams = Map[String, Object](
  // the usual params, make sure to change the port in bootstrap.servers if 9092 is not TLS
  "security.protocol" -> "SSL",
  "ssl.truststore.location" -> "/some-directory/kafka.client.truststore.jks",
  "ssl.truststore.password" -> "test1234",
  "ssl.keystore.location" -> "/some-directory/kafka.client.keystore.jks",
  "ssl.keystore.password" -> "test1234",
  "ssl.key.password" -> "test1234"
)
```

Deploying

As with any Spark applications, spark-submit is used to launch your application.

For Scala and Java applications, if you are using SBT or Maven for project management, then package spark-streaming-kafka-0-10_2.12 and its dependencies into the application JAR. Make sure spark-core_2.12 and spark-streaming_2.12 are marked as provided dependencies as those are already present in a Spark installation. Then use spark-submit to launch your application (see Deploying section in the main programming guide).

40.79.78.1

来自 <https://spark.apache.org/docs/latest/streaming-kafka-0-10-integration.html>

大数据学习之路97-kafka直连方式（spark streaming整合kafka 0.10版本）

2018年10月15日 11:10:08 爱米酱 阅读数 794

我们之前SparkStreaming整合Kafka的时候用的是傻瓜式的方式-----createStream，但是这种方式的效率很低。而且在kafka 0.10版本之后就不再提供了。

接下来我们使用Kafka直连的方式，这种方式其实是调用Kafka底层的消费数据的API，我们知道，越底层的东西效率也就越高。

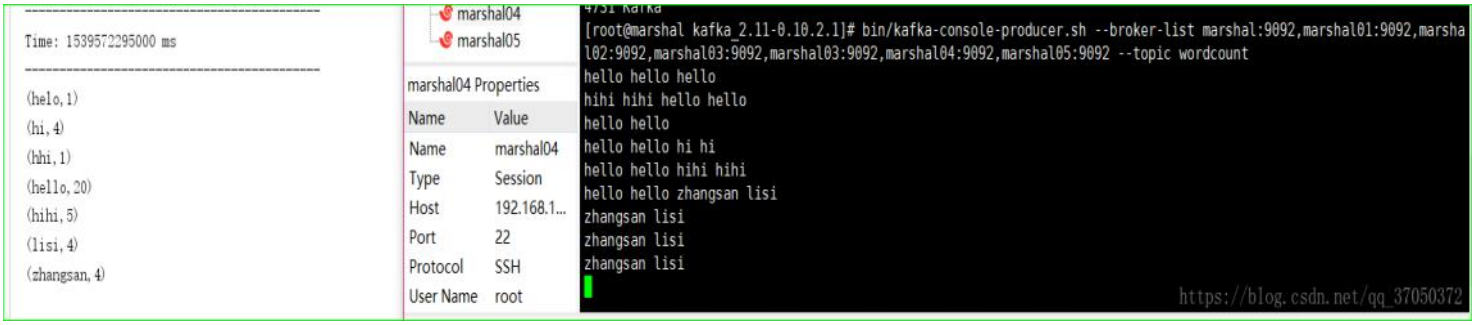
使用之前的方式是要连接到zookeeper的，而现在的方式则不需要。

代码如下：

```
package com.test.sparkStreaming
import org.apache.kafka.clients.consumer.ConsumerRecord
import org.apache.kafka.common.serialization.StringDeserializer
import org.apache.spark.{HashPartitioner, SparkConf}
import org.apache.spark.rdd.RDD
import org.apache.spark.streaming.dstream.{DStream, InputDStream}
import org.apache.spark.streaming.kafka010._
import org.apache.spark.streaming.{Seconds, StreamingContext}
object KafkaDirectStream {
val updateFunc = (it : Iterator[(String, Seq[Int], Option[Int])]) => {
it.map{case (w,s,o) => (w,s.sum + o.getOrElse(0))}
}
def main(args: Array[String]): Unit = {
val conf: SparkConf = new
SparkConf().setAppName("KafkaDirectStream").setMaster("local[*]")
val ssc: StreamingContext = new StreamingContext(conf,Seconds(5))
ssc.checkpoint("./ck")
//跟Kafka整合（直连方式，调用Kafka底层的消费数据的API）
val brokerList =
"marsha1:9092,marsha101:9092,marsha102:9092,marsha103:9092,marsha104:9092,marsha105:9
092"
val kafkaParams = Map[String,Object](
"bootstrap.servers" -> brokerList,
"key.deserializer" -> classOf[StringDeserializer],
"value.deserializer" -> classOf[StringDeserializer],
"group.id" -> "g100",
//这个代表，任务启动之前产生的数据也要读
"auto.offset.reset" -> "earliest",
"enable.auto.commit" -> (false:java.lang.Boolean)
)
val topics = Array("wordcount")
/**
*  指定kafka数据源
*  ssc：StreamingContext的实例
*  LocationStrategies：位置策略，如果kafka的broker节点跟Executor在同一台机器上给一种
策略，不在一台机器上给另外一种策略
*      设定策略后会以最优的策略进行获取数据
*      一般在企业中kafka节点跟Executor不会放到一台机器的，原因是kakfa是消息存储的，
Executor用来做消息的计算，
*      因此计算与存储分开，存储对磁盘要求高，计算对内存、CPU要求高
*      如果Executor节点跟Broker节点在一起的话使用PreferBrokers策略，如果不在一起的话
使用PreferConsistent策略
*      使用PreferConsistent策略的话，将来在kafka中拉取了数据以后尽量将数据分散到所有
的Executor上
*  ConsumerStrategies：消费者策略（指定如何消费）
*
*/
val directStream: InputDStream[ConsumerRecord[String, String]] =
KafkaUtils.createDirectStream(ssc,
LocationStrategies.PreferConsistent,
ConsumerStrategies.Subscribe[String,String](topics,kafkaParams)
)
val result: DStream[(String, Int)] = directStream.map(_.value()).flatMap(_.split("
"))
.map((_, 1))
.updateStateByKey(updateFunc, new
HashPartitioner(ssc.sparkContext.defaultParallelism), true)
result.print()
directStream.foreachRDD(rdd => {
val offsetRange: Array[OffsetRange] = rdd.asInstanceOf[HasOffsetRanges].offsetRanges
val maped: RDD[(String, String)] = rdd.map(record =>(record.key,record.value))
```

```
//计算逻辑
maped.foreach(println)
//循环输出
for(o<-offsetRange){
println(s"${o.topic} ${o.partition} ${o.fromOffset} ${o.untilOffset}")
}
})
ssc.start()
ssc.awaitTermination()
}
}
```

运行结果如下：



来自 <https://blog.csdn.net/qq_37050372/article/details/83052951>