概念

- **topic**: Kafka维护消息类别的东西是主题
- **producer**: 我们称发布消息到Kafka主题的进程叫生产者
- **consumer**: 我们称订阅主题、获取消息的进程叫消费者
- **broker**: Kafka是由多个服务器组成的机器，每个服务器称作代理
  在较高的层次上看，生产者通过网络发送消息到Kafka集群，Kafka集群将这些消息提供给消费者。
  客户端与服务器之间的通信通过一个简单的、高性能的、语言无关的TCP protocol。Kafka有Java客户端，但这客户端在很多语言many languages也是有效的。
- **分区**：单个Topic被切分成多个分区log，分区文件可以分布在多个broker上。分区数目越多，意味着消费者并发处理记录的能力越强。

日志文件分区

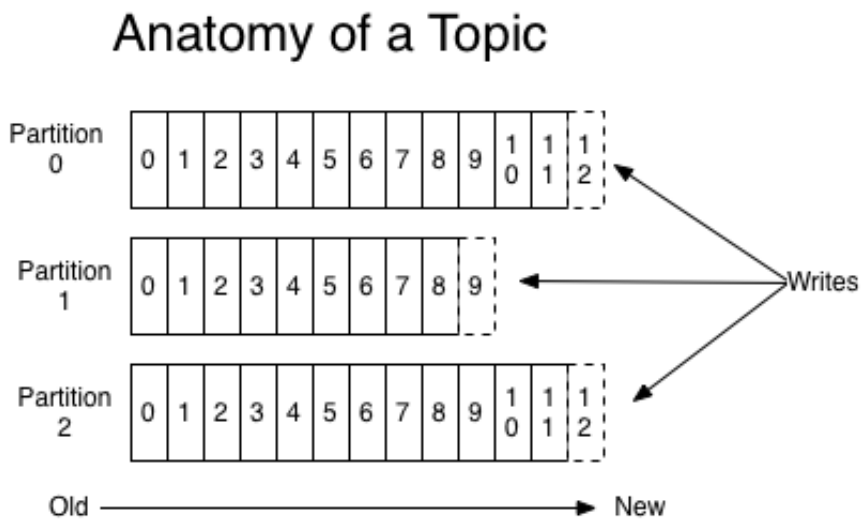
一个主题就是消息的类别或名称。对每个主题，Kafka集群都管理着一个被分区的日志。

每个topic 将被分成多个partition(区)，每个partition 在存储层面是append log 文件。任何发布到此partition 的消息都会被直接追加到log 文件的尾部，追加的log有固定的结构（称为commit log）。

Producer 将消息发布到指定的Topic中。**同时Producer 也能决定将此消息归属于哪个partition**（这可以通过简单的循环的方式来实现，或者使用一些分区方法）。

Offset

每条消息在文件中的位置称为**offset（偏移量）**，**offset 为一个long型数字，它是唯一标记一条消息。**



kafka 并没有提供其他额外的索引机制来存储offset，因为在kafka 中几乎不允许对消息进行“随机读写”。

对于consumer 而言，它需要保存消费消息的offset，对于offset的保存和使用, 由consumer 来控制； 当consumer 正常消费消息时，offset 将会"线性"的向前驱动,即消息将依次顺序被消费。事实上consumer 可以使用任意顺序消费消息,它只需要将offset重置为任意值。(offset 将会保存在zookeeper 中,参见下文)。

备份因子

kafka 还可以配置partitions 需要备份的个数(replicas)，每个partition 将会被备份到多台机器上，以提高可用性。

基于replicated（冗余）方案，那么就意味着需要对多个备份进行调度;每个partition都有一个机器为"leader"；零个或多个机器作为follower。

leader 负责所有的读写操作，follower执行leader的指令。如果leader 失效，那么将会有其他follower 来接管(成为新的leader)；

follower只是单调的和leader 跟进,同步消息即可。由此可见作为leader 的server 承载了全部的请求压力，因此从集群的整体考虑,有多少个partitions就意味着有多少个"leader"，kafka会将"leader"均衡的分散在每个实例上，来确保整体的性能稳定。

消费者

传统的消息传递有两种方式： 队列方式（queuing）、发布-订阅（publish-subscribe）方式。

- 队列方式：一组消费者从机器上读消息，每个消息只传递给这组消费者中的一

个。

- 分布-订阅方式：消息被广播到所有的消费者。
  Kafka提供了一个消费组（consumer group）的说法来概括这两种方式。**只要消费组订阅了Topic就能收到消息（一个消息可以被多个消费组订阅），但是消息只能被消费组中的一个消费者使用。**
- **队列方式：所有消费者属于同一个消费组；**
- **消息广播：所有消费者属于不同的消费组；**

kafka只能保证一个partition中的消息被某个consumer 消费时，消息是顺序的。事实上,从Topic 角度来说，消息仍不是有序的（一个topic可能有多个分区）。

在kafka中消费记录的方式是：**将日志中的分区划分到消费者实例上**，因此对消费者来说，任何时间点都会独占Topic的一个或者多个分区（Kafka的协议会动态调整这个分区分配策略）。----- 按照各个说法，同一个时间内被消费的record数目，取决于min**（消费者数目，分区数目）**

MirrorMaker

一个用来跨集群同步的工具，使用MirrorMaker可以在两个Kafka集群之间同步Topic数据。

多租户

允许对消费者和生产者进行权限校验，参考

Kafka Manager安装

Parcel包

当前Cloudera官方没有提供相关Parcel包，仅仅只有第三方的制作的Parcel包。包的版本只到1.2.7，且不支持Centos7以上的系统。

Docker

也是第三方制作的镜像，版本也是1.2.7：

```
docker pull sheepkiller/kafka-manager
```

```
# 注意这里不能用来主机名来指定ZK地址
 docker run -it -d --rm  -p 9000:9000 --name kafka-manager \
-e ZK_HOSTS="172.24.10.78:2181,172.24.10.67:2181,172.24.10.34:2181" \
-e APPLICATION_SECRET=letmein \
sheepkiller/kafka-manager
```

从源码安装

```
# GitHub下载源码https://github.com/yahoo/kafka-manager.git
# 解压进入目录执行（需要安装JDK8，并且配置JAVA_HOME,不太清楚是不是要装sbt，好像
./sbt clean dist
# 打成rpm包
sbt rpm:packageBin
# 修改配置文件conf/application.conf,配置ZK地址
kafka-manager.zkhosts="lqbdnode2:2181,lqbdnode1:2181,lqbdnode3:2181"
# 启动服务
nohup bin/kafka-manager -Dconfig.file=conf/application.conf -Dhttp.port=18
```

官方Quick Start

```
# 创建主题
kafka-topics --create --zookeeper lqbdnode1:2181,lqbdnode2:2181,lqbdnode3:
# 列出所有主题
kafka-topics --list --zookeeper lqbdnode1:2181,lqbdnode2:2181,lqbdnode3:21
# 通过控制台向指定topic发送数据，注意9092是broker的监听端口
kafka-console-producer --broker-list localhost:9092 --topic test
# 从控制台启动一个消费者
kafka-console-consumer --bootstrap-server localhost:9092 --topic test --fr
# 删除一个Topic，（当配置项delete.topic.enable为True时，才能用工具删除）
kafka-topics --delete --zookeeper lqbdnode1:2181 --topic test
# 打印Topic信息
kafka-topics --describe --zookeeper localhost:2181 --topic my-replicated-t
```

配置多个Broker

```
# 多个Broker可以配置在同一台机器上，但是要保证他们的log目录、broker.id、TCP端口
broker.id=1    # 用来标识集群的唯一ID
listeners=PLAINTEXT://:9092 # TCP端口号
log.dir=/tmp/kafka-logs-1 # log存储目录
```

USE CASES

- 消息队列
- Website Activity Tracking
- 量测数据监控
- Log Aggregation
- Event Sourcing、Commit Log
- 流处理

配置项

参考

API说明

Kafka有4个核心API：

- Producer API 允许应用程序发送数据流到kafka集群中的topic。
- Consumer API 允许应用程序从kafka集群的topic中读取数据流。
- Streams API 允许从输入topic转换数据流到输出topic。
- Connect API 通过实现连接器（connector），不断地从一些源系统或应用程序中拉取数据到kafka，或从kafka提交数据到宿系统（sink system）或应用程序。

kafka公开了其所有的功能协议，与语言无关。只有java客户端作为kafka项目的一部分进行维护，其他的作为开源的项目提供（Kafka第三方API接口）。

生产者客户端

KafkaProducer

- 线程安全的，当多个线程共用一个KafkaProducer时，有更高的效率；
- KafkaProducer对象为发往每个分区的记录配置了指定的缓存空间;
- send()接口是异步接口，当调用该接口后会立即返回。此时记录被追加到了客户端的Producer缓存中，Producer单记录积累到一定数量时，一次性发送；

获取send的返回值

``` java
// 通过send方法的返回值Future,查看消息的分区，offset --- 但是这样会阻塞
RecordMetadata res = producer.send(
new ProducerRecord("hellp", "LinQing", "Hello World")
).get();
System.out.println(res.partition());
System.out.println(res.offset());
// 通过回调函数
// 可以保证，回调函数的执行顺序和发送消息的顺序一致
producer.send(new ProducerRecord("hellp", "dfadf", "asdfa"),new Callback() {
public void onCompletion(RecordMetadata metadata, Exception e) {
if (e != null)
e.printStackTrace();
System.out.println("The offset of the record we just sent is: " + metadata.offset());
}
});
```

配置项

可以参考类

- org.apache.kafka.clients.producer.ProducerConfig

参数	说明
linger.ms	缓存停留时间
buffer.memory	缓存的大小
batch.size	缓存的记录数目
acks	缓冲消息发送成功的标志，all表示当缓冲满时,继续发送消息会造成阻塞
retries	表示发送失败时允许重试,重试可能造成消息重复
enable.idempotence	启用idempotent producer能力，该特性开启后，加强了消息 exactly once delivery语义，保证多次重试，也不会造成重复消息。配置该配置为true时，retries默认为 Integer.MAX_VALUE，acks默认为ALL
transactional.id	启用Kafka的事务处理能力，消息只有被消费之后才算被发送？？启用该配置之后，consumer也要相应配置，并且会导致Producer变成同步的。一个Session只能启用一个事务，并且所有发送操作应该在 `beginTransaction()` 和 `commitTransaction()` 之间。

如何发送到指定的分区

发送消息到topic的哪个分区取决于ProducerRecord是如何创建的：

```
ProducerRecord(java.lang.String topic,
               java.lang.Integer partition,
               java.lang.Long timestamp,
```

```
                K key, V value)
    /**
    topic：指定发送的topic
    partition：指定发送的分区（最高优先级）
    timestamp：时间戳，默认为当前时间，取决于topic的具体配置
    key：当没有指定partition时，发送到哪个分区，取决于key的hash值，partition和
    value：发送的消息
    **/

```

消费者客户端

KafkaConsumer

- 自动处理连接brokers时的失败，当topic发生分区迁移时，也能自动处理；
- KafkaConsumer和KafkaProducer不同，他不是线程安全的；
- 使用同一个group.id的Consumer属于同一个消费者组，消费者组可以订阅多个Topic，并且能够动态改变；
- 分区和消费者组的对应关系是kafka动态调整的；
- Topic重新分区时，消费者可以通过`ConsumerRebalanceListener，`获取通知；
- 消费者可以通过`assign(Collection)`进行手动分区分配，这种情况下Consumer可以只接收，自己关心的分区消息（此时要使用一个独立的group.id防止冲突）；
- `subscribe`接口订阅消息时是自动分配分区的；

控制消息的偏移位置

Kafka允许将offset保存起来，使消费者恢复时能够继续未完成的工作，需要注意以下几点：

- `enable.auto.commit=false`
- 手工为每个分区调用commitSync
- 使用`seek(TopicPartition, long)`定位消息位置
- 要注意处理分区重新分配的场景；

手动调用commit

```java
// 手动commit的例子
ConsumerRecords records = consumer.poll(Long.MAX_VALUE);
for (TopicPartition partition : records.partitions()) {
List<ConsumerRecord> partitionRecords = records.records(partition);
for (ConsumerRecord record : partitionRecords) {
System.out.println(record.offset() + ": " + record.value());
}
long lastOffset = partitionRecords.get(partitionRecords.size() - 1).offset();
consumer.commitSync(Collections.singletonMap(partition, new
OffsetAndMetadata(lastOffset + 1)));
}
```

重定位到开头或者结尾，默认情况下，一个consumer加入时似乎offset在最末端：

```
seekToBeginning(Collection)
seekToEnd(Collection))
```

控制从哪个分区获取message：

```
pause(Collection):调用pause后下次poll返回的record不包含指定的分区；
resume(Collection)：调用resume后下次poll返回的record包含指定的分区；

```

Reading Transactional Messages

略

Multi-threaded Processing

```
```

```java
/***
* KafkaConsumer不是线程安全的，当一个外部线程要使用其他线程的KafkaConsumer时需
* 使用wakeup接口可以关闭一个KafkaConsumer，工作线程会抛出WakeupException
*/

class KafkaConsumerRunner implements Runnable {
    private final AtomicBoolean closed = new AtomicBoolean(false);
    private final KafkaConsumer consumer =null;
    public void run() {
        try {
            consumer.subscribe(Arrays.asList("topic"));
            while (!closed.get()) {
                ConsumerRecords records = consumer.poll(10000);
                // Handle new records
            }
        } catch (WakeupException e) {
            // Ignore exception if closing
            if (!closed.get()) throw e;
        } finally {
            consumer.close();
        }
    }
    // Shutdown hook which can be called from a separate thread
    public void shutdown() {
        closed.set(true);
        consumer.wakeup();
    }
}
```

分区重平衡

consumer需要频繁的调用 `poll(long)` 来保证自己存活，如果consumer超过 `session.timeout.ms` 没有发送心跳，kafka会将consumer从group中移除，并且重新平衡分区。

`max.poll.interval.ms` 参数要求，consumer调用poll的频率小于这个值，否则会自动退出group。

Kafka Connector

Kafka Connector是能够将其他系统和Kafka快速集成的框架。

通过Connnector可以将外部数据导入到topic中，或者将Topic的数据快速导入到外部系统。

- Kafka连接器通用框架：Kafka Connect 规范了kafka与其他数据系统集成的框架，利用这个框架可以快速的桥接Kafka和其他系统。
- 分布式和单机模式：可以单机或者分布式部署Kafka Connector
- REST 接口 ：能够使用REST API管理Connector集群
- 自动offset管理：无需费心管理offset
- Kafka安装包提供了一些开源的Connector，配置文件在conf目录，如何使用可以参考官网。
- confluent会提供一些商业的Connector

Kafka Streams

参考官网

Kafka 对接 Spark Streaming

参考官方

方式一：Receiver-based Approach

这种方式在故障时，可能会丢失数据，否则需要Spark开启Write Ahead Logs对Kafka的数据进行持久化。

这种方式，使用的是Comsumer API（Receiver），有下面特点：

- ssc的rdd分区和kafka的topic分区不是一个概念，增加Topic分区数仅仅是增加一个Receiver中消费topic的线程数，并不增加spark的并行处理数据数量 ； ------ 两个分区完全没有直接关系
- Kafka会在ZK上维护Offset ---- 主要是为了一些Monitor工具；
- 如果要增加Receiver并行度，需要创建多个DStream分别接受不同Top或者分区的消息

```
// 在Spark2中已经找不到这个接口
import org.apache.spark.streaming.kafka._
val kafkaStream = KafkaUtils.createStream(streamingContext,[ZK quorum], [c
```

方式二：Direct Approach

这种方式并不在Executor启动一个Receiver接受所有的Record，而是定期查询分区的lastoffset，Executor根据lastoffset划分出每个batch需要处理的range，job进程根据range获取kafka上的数据。

这种方式的特点：

- RDD的分区数和Topic的分区数时一致的；
- 和上面的方式相比更有效率，主要原因是无需写日志，当数据丢失时job可以从kafka重新获取；
- 不会在ZK上维护offset信息（原因是不使用消费者API，而是调用更底层的API）；
- 实现“恰好一次”语义，参考。（原因是传统方式中，可能出现SSC中的偏移和ZK中不一致）；

```
import org.apache.spark.streaming.kafka._
val directKafkaStream = KafkaUtils.createDirectStream[[key class], [value
```