

第1章 初识Kafka

tips 学完这一章你可以

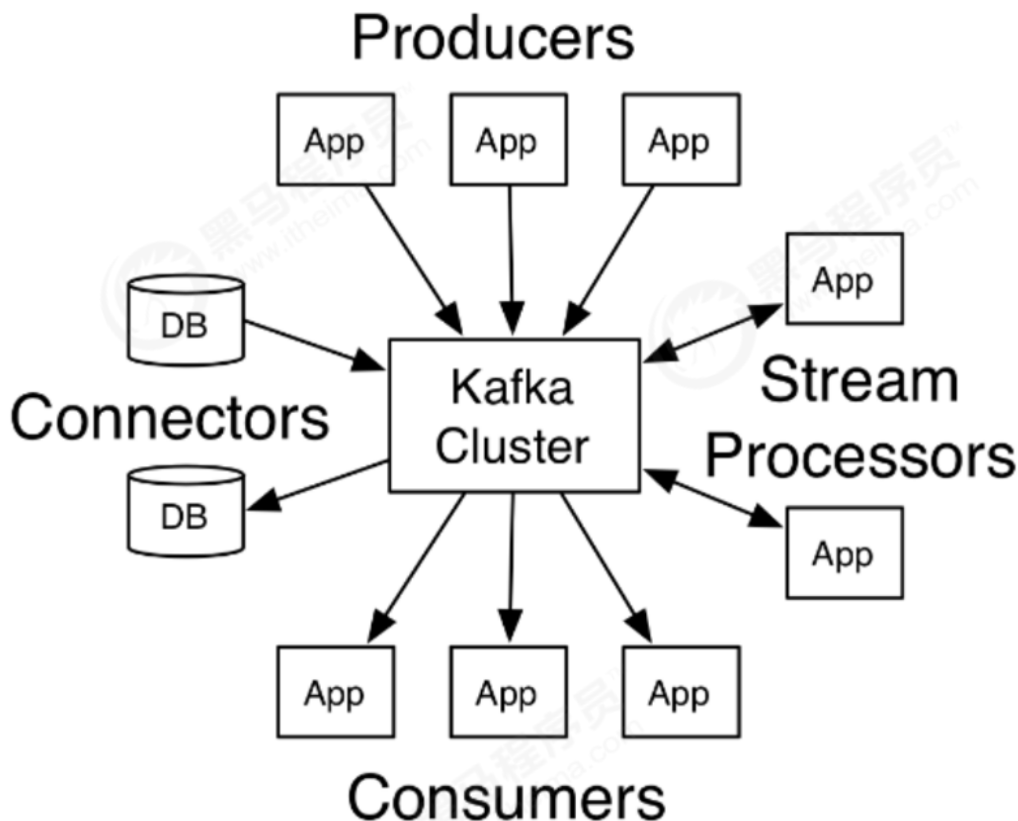
知道Kafka基本原理，了解关键术语概念

可以使用Kafka进行消息系统开发

通过Java语言来使用Kafka进行消息收发

Kafka最初是由LinkedIn公司采用Scala语言开发的一个多分区、多副本并且基于ZooKeeper协调的分布式消息系统，现在已经捐献给Apache基金会。目前Kafka已经定位为一个分布式流式处理平台，它以高吞吐、可持久化、可水平扩展、支持流处理等多种特性而被广泛应用。

Apache Kafka是一个分布式的发布-订阅消息系统，能够支撑海量数据的数据传递。在离线和实时的消息处理业务系统中，Kafka都有广泛的应用。Kafka将消息持久化到磁盘中，并对消息创建了备份保证了数据的安全。Kafka在保证较高的处理速度的同时，又能保证数据处理的低延迟和数据的零丢失。



特性

(1) 高吞吐量、低延迟：kafka每秒可以处理几十万条消息，它的延迟最低只有几毫秒，每个主题可以分多个分区，消费组对分区进行消费操作；

(2) 可扩展性：kafka集群支持热扩展；

(3) 持久性、可靠性：消息被持久化到本地磁盘，并且支持数据备份防止数据丢失；

(4) 容错性：允许集群中节点失败（若副本数量为n，则允许n-1个节点失败）；

(5) 高并发：支持数千个客户端同时读写；

北京市昌平区建材城西路金燕龙办公楼一层 电话：400-618-9090

(1) 日志收集：一个公司可以用Kafka可以收集各种服务的log，通过kafka以统一接口服务的方式开放给各种consumer，例如Hadoop、Hbase、Solr等；

(2) 消息系统：解耦和生产者和消费者、缓存消息等；

(3) 用户活动跟踪：Kafka经常被用来记录web用户或者app用户的各种活动，如浏览网页、搜索、点击等活动，这些活动信息被各个服务器发布到kafka的topic中，然后订阅者通过订阅这些topic来做实时的监控分析，或者装载到Hadoop、数据仓库中做离线分析和挖掘；

(4) 运营指标：Kafka也经常用来记录运营监控数据。包括收集各种分布式应用的数据，生产各种操作的集中反馈，比如报警和报告；

(5) 流式处理：比如spark streaming和storm；

技术优势

可伸缩性：Kafka 的两个重要特性造就了它的可伸缩性。

1、Kafka 集群在运行期间可以轻松地扩展或收缩（可以添加或删除代理），而不会宕机。

2、可以扩展一个 Kafka 主题来包含更多的分区。由于一个分区无法扩展到多个代理，所以它的容量受到代理磁盘空间的限制。能够增加分区和代理的数量意味着单个主题可以存储的数据量是没有限制的。

容错性和可靠性：Kafka 的设计方式使某个代理的故障能够被集群中的其他代理检测到。由于每个主题都可以在多个代理上复制，所以集群可以在不中断服务的情况下从此类故障中恢复并继续运行。

吞吐量：代理能够以超快的速度有效地存储和检索数据。

适应人群

本教程为专注于使用Apache Kafka消息传递系统或者大数据分析领域发展事业的专业人士做好准备，它将给你足够的理解如何使用Kafka集群。

课程亮点

l 知识覆盖度广泛；

l 知识覆盖度深入；

l 由浅入深讲解思路；

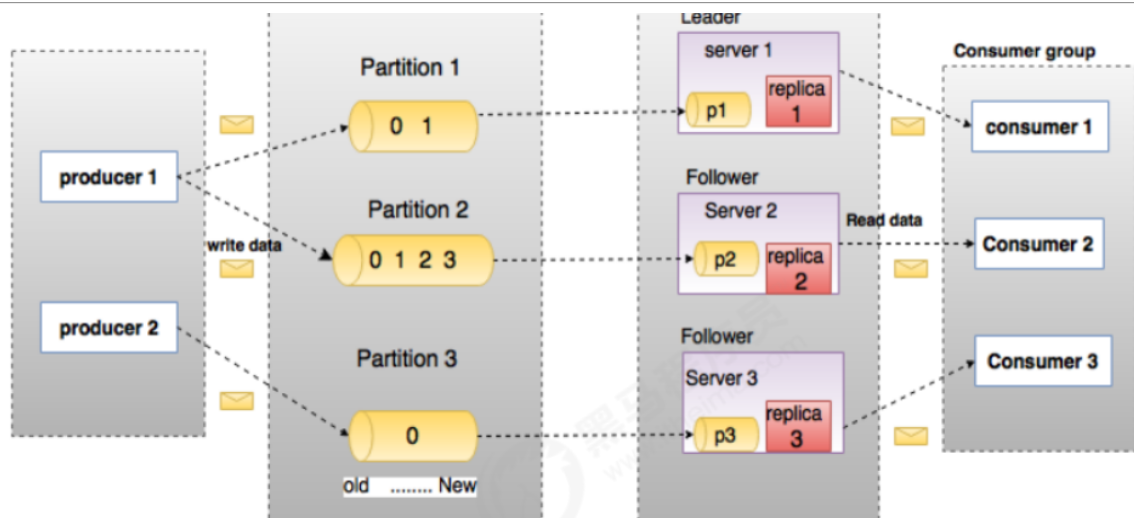
l 案例分析全面；

l 适应于想学习Kafka技术的不同人群；

Apache官网：<http://apache.org>

Kafka官网：<http://kafka.apache.org>

1.1 概念详解



Producer

生产者即数据的发布者，该角色将消息发布到Kafka的topic中。broker接收到生产者发送的消息后，broker将该消息追加到当前用于追加数据的segment文件中。生产者发送的消息，存储到一个partition中，生产者也可以指定数据存储的partition。

Consumer

消费者可以从broker中读取数据。消费者可以消费多个topic中的数据。

Topic

在Kafka中，使用一个类别属性来划分数据的所属类，划分数据的这个类称为topic。如果把Kafka看做一个数据库，topic可以理解为数据库中的一张表，topic的名字即为表名。

Partition

topic中的数据分割为一个或多个partition。每个topic至少有一个partition。每个partition中的数据使用多个segment文件存储。partition中的数据是有序的，partition间的数据丢失了数据的顺序。如果topic有多个partition，消费数据时就不能保证数据的顺序。在需要严格保证消息的消费顺序的场景下，需要将partition数目设为1。

Partition offset

每条消息都有一个当前Partition下唯一的64字节的offset，它指明了这条消息的起始位置。

Replicas of partition

副本是一个分区的备份。副本不会被消费者消费，副本只用于防止数据丢失，即消费者不从为follower的partition中消费数据，而是从为leader的partition中读取数据。副本之间是一主多从的关系。

Broker

Kafka 集群包含一个或多个服务器，服务器节点称为broker。broker存储topic的数据。如果某topic有N个partition，集群有N个broker，那么每个broker存储该topic的一个partition。如果某topic有N个partition，集群有(N+M)个broker，那么其中有N个broker存储该topic的一个partition，剩下的M个broker不存储该topic的partition数据。如果某topic有N个partition，集群中broker数目少于N个，那么一个broker存储该topic的一个或多个partition。在实际生产环境中，尽量避免这种情况的发生，这种情况容易导致Kafka集群数据不均衡。

Leader

Follower

Follower跟随Leader，所有写请求都通过Leader路由，数据变更会广播给所有Follower，Follower与Leader保持数据同步。如果Leader失效，则从Follower中选举出一个新的Leader。当Follower与Leader挂掉、卡住或者同步太慢，leader会把这个follower从“in sync replicas”（ISR）列表中删除，重新创建一个Follower。

Zookeeper

Zookeeper负责维护和协调broker。当Kafka系统中新增了broker或者某个broker发生故障失效时，由ZooKeeper通知生产者和消费者。生产者和消费者依据Zookeeper的broker状态信息与broker协调数据的发布和订阅任务。

AR(Assigned Replicas)

分区中所有的副本统称为AR。

ISR(In-Sync Replicas)

所有与Leader部分保持一定程度的副（包括Leader副本在内）本组成ISR。

OSR(Out-of-Sync-Replicas)

与Leader副本同步滞后过多的副本。

HW(High Watermark)

高水位，标识了一个特定的offset，消费者只能拉取到这个offset之前的消息。

LEO(Log End Offset)

即日志末端位移(log end offset)，记录了该副本底层日志(log)中下一条消息的位移值。注意是下一条消息！也就是说，如果LEO=10，那么表示该副本保存了10条消息，位移值范围是[0, 9]。



1.2 安装与配置

1.2.1 java环境

我们使用Linux系统进行教学演示，通过虚拟机安装CentOS或者Win10系统自己挂载的ubuntu系统都可以。

首先需要安装Java环境，同时配置环境变量，步骤如下：

- 官网下载DK

<https://www.oracle.com/technetwork/java/javase/downloads/jdk12-downloads-5295953.html>

You must accept the **Oracle Technology Network License Agreement for Oracle Java SE** to download this software.

Thank you for accepting the Oracle Technology Network License Agreement for Oracle Java SE; you may now download this software.

Product / File Description	File Size	Download
Linux	155.14 MB	jdk-12.0.2_linux-x64_bin.deb
Linux	162.79 MB	jdk-12.0.2_linux-x64_bin.rpm
Linux	181.68 MB	jdk-12.0.2_linux-x64_bin.tar.gz
macOS	173.63 MB	jdk-12.0.2_osx-x64_bin.dmg
macOS	173.98 MB	jdk-12.0.2_osx-x64_bin.tar.gz
Windows	158.63 MB	jdk-12.0.2_windows-x64_bin.exe
Windows	179.57 MB	jdk-12.0.2_windows-x64_bin.zip

- 解压缩,放到指定目录
- 配置环境变量

在/etc/profile文件中配置如下变量

```
export JAVA_HOME=/java/jdk-12.0.1
export JRE_HOME=$JAVA_HOME/jre
export CLASSPATH=.:$JAVA_HOME/lib:$JRE_HOME/lib:$CLASSPATH
export PATH=.:$JAVA_HOME/bin:$JRE_HOME/bin:$KE_HOME/bin:${MAVEN_HOME}/bin:$PATH
```

- 测试jdk

```
itcast@Server-node:~$ which java
/java/jdk-12.0.1/bin/java
itcast@Server-node:~$ java -version
java version "12.0.1" 2019-04-16
Java(TM) SE Runtime Environment (build 12.0.1+12)
Java HotSpot(TM) 64-Bit Server VM (build 12.0.1+12, mixed mode, sharing)
```

1.2.2 ZooKeeper的安装

Zookeeper是安装Kafka集群的必要组件，Kafka通过Zookeeper来实施对元数据信息的管理，包括集群、主题、分区等内容。

同样在官网下载安装包到指定目录解压缩，步骤如下：

- ZooKeeper官网：<http://zookeeper.apache.org/>

```
itcast@Server-node:/mnt/d/zookeeper-3.4.14$ ls
LICENSE.txt      conf              src               zookeeper-client  zookeeper-server
NOTICE.txt       dist-maven       test             zookeeper-control zookeeper.out
README.md        ivy.xml          zookeeper-3.4.14.jar  zookeeper-docs    zookeeper-ll
README_packaging.txt  ivysettings.xml zookeeper-3.4.14.jar.asc  zookeeper-ll      zookeeper-jute
bin              cli              zookeeper-3.4.14.jar.md5  zookeeper-jute    zookeeper-recipes
build.xml         pom.xml          zookeeper-3.4.14.jar.sha1
```

- 修改Zookeeper的配置文件，首先进入安装路径conf目录，并将zoo_sample.cfg文件修改为zoo.cfg，并对核心参数进行配置。

文件内容如下：

```
# The number of milliseconds of each tick
# zk服务器的心跳时间
tickTime=2000
# The number of ticks that the initial
# synchronization phase can take
# 投票选举新Leader的初始化时间
initLimit=10
# The number of ticks that can pass between
# sending a request and getting an acknowledgement
```

北京市昌平区建材城西路金燕龙办公楼一层 电话：400-618-9090



```
# do not use /tmp for storage, /tmp here is just
# example sake.
# 数据目录
dataDir=temp/zookeeper/data
# 日志目录
dataLogDir=temp/zookeeper/log
# the port at which the clients will connect
# Zookeeper对外服务端口，保持默认
clientPort=2181
```

- 启动Zookeeper命令：bin/zkServer.sh start

```
itcast@Server-node:/mnt/d/zookeeper-3.4.14$ bin/zkServer.sh start
Zookeeper JMX enabled by default
Using config: /mnt/d/zookeeper-3.4.14/bin/../conf/zoo.cfg
//启动成功
Starting zookeeper ... STARTED
itcast@Server-node:/mnt/d/zookeeper-3.4.14$
```

1.2.3 Kafka的安装与配置

- 官网下载安装解压缩：<http://kafka.apache.org/downloads>

DOCUMENTATION	2.3.0
PERFORMANCE	<ul style="list-style-type: none"> Released Jun 25, 2019 Release Notes
POWERED BY	<ul style="list-style-type: none"> Source download: kafka-2.3.0-src.tgz (asc, sha512)
PROJECT INFO	<ul style="list-style-type: none"> Binary downloads: <ul style="list-style-type: none"> Scala 2.11 - kafka_2.11-2.3.0.tgz (asc, sha512) Scala 2.12 - kafka_2.12-2.3.0.tgz (asc, sha512)
ECOSYSTEM	

- 下载解压启动

启动命令：bin/kafka-server-start.sh config/server.properties

server.properties配置中需要关注以下几个参数：

broker.id=0 表示broker的编号，如果集群中有多个broker，则每个broker的编号需要设置的不同

listeners=PLAINTEXT://:9092 broker对外提供的服务入口地址

log.dirs=/tmp/kafka/log 设置存放消息日志文件的地址

zookeeper.connect=localhost:2181 Kafka所需Zookeeper集群地址，教学中Zookeeper和Kafka都安装本机

- 启动成功如下显示：

```
INFO [SocketServer brokerId=0] Started data-plane processors for 1 acceptors (kafka.network.Sc
INFO Kafka version: 2.2.1 (org.apache.kafka.common.utils.AppInfoParser)
INFO Kafka commitId: 55783d3133a5a49a (org.apache.kafka.common.utils.AppInfoParser)
INFO [KafkaServer id=0] started (kafka.server.KafkaServer)
```

- 启动成功之后重新打开一个终端，验证启动进程


```
erties -cp ./java/jdk-12.0.1/lib:/java/jdk-12.0.1/jre/lib:/java/jdk-12.0.1/lib:/java/jdk-12.0.1/jre/lib:/java/jdk-12.0.1/lib:/java
.1/jre/lib:/mnt/d/kafka_2.12-2.2.1/bin/./libs/activation-1.1.1.jar:/mnt/d/kafka_2.12-2.2.1/bin/./libs/aopalliance-repackaged-2.5.0-b4
t/d/kafka_2.12-2.2.1/bin/./libs/argparse4j-0.7.0.jar:/mnt/d/kafka_2.12-2.2.1/bin/./libs/audience-annotations-0.5.0.jar:/mnt/d/kafka_2.
bin/./libs/commons-lang3-3.8.1.jar:/mnt/d/kafka_2.12-2.2.1/bin/./libs/connect-api-2.2.1.jar:/mnt/d/kafka_2.12-2.2.1/bin/./libs/connec
uth-extension-2.2.1.jar:/mnt/d/kafka_2.12-2.2.1/bin/./libs/connect-file-2.2.1.jar:/mnt/d/kafka_2.12-2.2.1/bin/./libs/connect-json-2.2.
t/d/kafka_2.12-2.2.1/bin/./libs/connect-runtime-2.2.1.jar:/mnt/d/kafka_2.12-2.2.1/bin/./libs/connect-transforms-2.2.1.jar:/mnt/d/kafka
```

```
itcast@Server-node:/mnt/d/kafka_2.12-2.2.1$ jps -l
868 jdk.jcmd/sun.tools.jps.Jps
472 kafka.Kafka
88 org.apache.zookeeper.server.quorum.QuorumPeerMain
```

1.2.4 Kafka测试消息生产与消费

- 首先创建一个主题

命令如下：

bin/kafka-topics.sh --zookeeper localhost:2181 --create --topic heima --partitions 2 --replication-factor 1

--zookeeper：指定了Kafka所连接的Zookeeper服务地址
--topic：指定了所要创建主题的名称
--partitions：指定了分区个数
--replication-factor：指定了副本因子
--create：创建主题的动作指令

```
itcast@Server-node:/mnt/d/kafka_2.12-2.2.1$ bin/kafka-topics.sh --zookeeper
localhost:2181 --create --topic heima --partitions 2 -
-replication-factor 1
//主题创建成功
Created topic heima.
itcast@Server-node:/mnt/d/kafka_2.12-2.2.1$
```

- 展示所有主题

命令：bin/kafka-topics.sh --zookeeper localhost:2181 --list

```
itcast@Server-node:/mnt/d/kafka_2.12-2.2.1$ bin/kafka-topics.sh --zookeeper
localhost:2181 --list
heima
itcast@Server-node:/mnt/d/kafka_2.12-2.2.1$
```

- 查看主题详情

命令：bin/kafka-topics.sh --zookeeper localhost:2181 --describe --topic heima

--describe 查看详情动作指令

```
itcast@Server-node:/mnt/d/kafka_2.12-2.2.1$ bin/kafka-topics.sh --zookeeper
localhost:2181 --describe --topic heima
Topic:heima      PartitionCount:2      ReplicationFactor:1      Configs:
Topic: heima     Partition: 0          Leader: 0                 Replicas: 0              Isr: 0
Topic: heima     Partition: 1          Leader: 0                 Replicas: 0              Isr: 0
itcast@Server-node:/mnt/d/kafka_2.12-2.2.1$
```

命令：bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic heima

--bootstrap-server 指定了连接Kafka集群的地址

--topic 指定了消费端订阅的主题

```
itcast@Server-node:/mnt/d/kafka_2.12-2.2.1$ bin/kafka-console-consumer.sh --
bootstrap-server localhost:9092 --topic heima
Hello,kafka!
```

- 生产端发送消息

命令：bin/kafka-console-producer.sh --broker-list localhost:9092 --topic heima

--broker-list 指定了连接的Kafka集群的地址

--topic 指定了发送消息时的主题

```
itcast@Server-node:/mnt/d/kafka_2.12-2.2.1$ bin/kafka-console-producer.sh --
broker-list localhost:9092 --topic heima
>Hello,kafka!
>
```

1.3 Java第一个程序

通过Java程序来进行Kafka收发消息的教学演示

1.3.1 准备

Kafka自身提供的Java客户端来演示消息的收发，与Kafka的Java客户端相关的Maven依赖如下：

```
<properties>
    <scala.version>2.11</scala.version>
    <slf4j.version>1.7.21</slf4j.version>
    <kafka.version>2.0.0</kafka.version>
    <lombok.version>1.18.8</lombok.version>
    <junit.version>4.11</junit.version>
    <gson.version>2.2.4</gson.version>
    <protobuf.version>1.5.4</protobuf.version>
    <spark.version>2.3.1</spark.version>
</properties>

<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>${kafka.version}</version>
</dependency>
```

1.3.2 创建生产者

见代码库 com.heima.kafka.chapter1.ProducerFastStart

```
/**
 * kafka 消息生产者
 * 北京市昌平区建材城西路金燕龙办公楼一层 电话：400-618-9090
```




```
// Kafka集群地址
private static final String brokerList = "localhost:9092";
// 主题名称-之前已经创建
private static final String topic = "heima";

public static void main(String[] args) {
    Properties properties = new Properties();
    // 设置key序列化器
    properties.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer");

    //另外一种写法
    //properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());

    // 设置重试次数
    properties.put(ProducerConfig.RETRIES_CONFIG, 10);
    // 设置值序列化器
    properties.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
    // 设置集群地址
    properties.put("bootstrap.servers", brokerList);
    // KafkaProducer 线程安全
    KafkaProducer<String, String> producer = new KafkaProducer<>
(properties);
    ProducerRecord<String, String> record = new ProducerRecord<>(topic,
"Kafka-demo-001", "hello, Kafka!");
    try {
        producer.send(record);
    } catch (Exception e) {
        e.printStackTrace();
    }
    producer.close();
}
}
```

1.3.3 创建消费者

见代码库 com.heimakafka.chapter1.ConsumerFastStart

```
/**
 * kafka 消息消费者
 */
public class ConsumerFastStart {
    // Kafka集群地址
    private static final String brokerList = "127.0.0.1:9092";
    // 主题名称-之前已经创建
    private static final String topic = "heima";
    // 消费组
    private static final String groupId = "group.demo";

    public static void main(String[] args) {
        Properties properties = new Properties();
        properties.put("key.deserializer",
            "org.apache.kafka.common.serialization.StringDeserializer");
    }
}
```

北京市昌平区建材城西路金燕龙办公楼一层 电话：400-618-9090

```
properties.put("bootstrap.servers", brokerList);
properties.put("group.id", groupId);

KafkaConsumer<String, String> consumer = new KafkaConsumer<>
(properties);
consumer.subscribe(Collections.singletonList(topic));

while (true) {
    ConsumerRecords<String, String> records =
        consumer.poll(Duration.ofMillis(1000));
    for (ConsumerRecord<String, String> record : records) {
        System.out.println(record.value());
    }
}
}
```

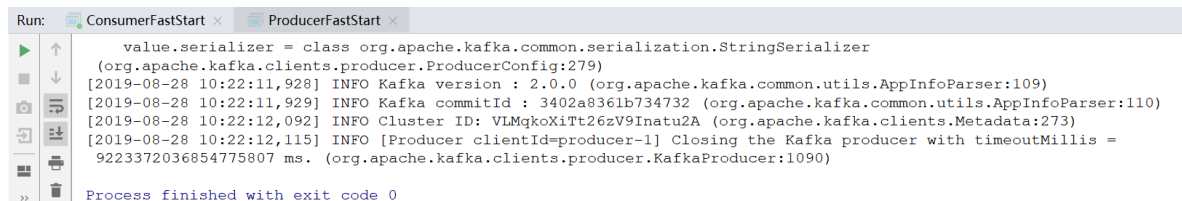
1.3.4 效果展示

waring：使用java连接linux下kafka集群需要设置hosts绑定；

先启动消费端，再启动生产端进行消息的发送

```
INFO Kafka version : 2.0.0 (org.apache.kafka.common.utils.AppInfoParser:109)
INFO Kafka commitId : 3402a8361b734732
(org.apache.kafka.common.utils.AppInfoParser:110)
INFO Cluster ID: VLMqkoXiTt26zV9Inatu2A (org.apache.kafka.clients.Metadata:273)
INFO [Producer clientId=producer-1] Closing the kafka producer with
timeoutMillis = 9223372036854775807 ms.
(org.apache.kafka.clients.producer.KafkaProducer:1090)
```

- 发送消息：



```
Run: ConsumerFastStart x ProducerFastStart x
value.serializer = class org.apache.kafka.common.serialization.StringSerializer
(org.apache.kafka.clients.producer.ProducerConfig:279)
[2019-08-28 10:22:11,928] INFO Kafka version : 2.0.0 (org.apache.kafka.common.utils.AppInfoParser:109)
[2019-08-28 10:22:11,929] INFO Kafka commitId : 3402a8361b734732 (org.apache.kafka.common.utils.AppInfoParser:110)
[2019-08-28 10:22:12,092] INFO Cluster ID: VLMqkoXiTt26zV9Inatu2A (org.apache.kafka.clients.Metadata:273)
[2019-08-28 10:22:12,115] INFO [Producer clientId=producer-1] Closing the Kafka producer with timeoutMillis =
9223372036854775807 ms. (org.apache.kafka.clients.producer.KafkaProducer:1090)
Process finished with exit code 0
```

- 订阅消息：



```
Run: ConsumerFastStart x ProducerFastStart x
hello, Kafka!
```

1.4 服务端常用参数配置

参数配置：config/server.properties

```
itcast@Server-node:/mnt/d/kafka_2.12-2.2.1$ egrep
'zookeeper|listeners|broker.id|log.dir|log.dirs' config/server.properties
broker.id=0
```



```
listeners=PLAINTEXT://:9092
# it uses the value for "listeners" if configured. Otherwise, it will use the
value
#advertised.listeners=PLAINTEXT://your.host.name:9092
#log.dirs=/tmp/kafka-logs
log.dirs=/tmp/kafka/log
# Zookeeper connection string (see zookeeper docs for details).
zookeeper.connect=localhost:2181
# Timeout in ms for connecting to zookeeper
zookeeper.connection.timeout.ms=6000
itcast@Server-node:/mnt/d/kafka_2.12-2.2.1$ egrep
'zookeeper|listeners|broker.id|log.dir|log.dirs|message.max'
config/server.properties
broker.id=0
#   listeners = listener_name://host_name:port
#   listeners = PLAINTEXT://your.host.name:9092
listeners=PLAINTEXT://:9092
# it uses the value for "listeners" if configured. Otherwise, it will use the
value
#advertised.listeners=PLAINTEXT://your.host.name:9092
#log.dirs=/tmp/kafka-logs
log.dirs=/tmp/kafka/log
# Zookeeper connection string (see zookeeper docs for details).
zookeeper.connect=localhost:2181
# Timeout in ms for connecting to zookeeper
zookeeper.connection.timeout.ms=6000
itcast@Server-node:/mnt/d/kafka_2.12-2.2.1$
```

1.4.1 zookeeper.connect

指明Zookeeper主机地址，如果zookeeper是集群则以逗号隔开，如：

172.6.14.61:2181,172.6.14.62:2181,172.6.14.63:2181

1.4.2 listeners

监听列表，broker对外提供服务时绑定的IP和端口。多个以逗号隔开，如果监听器名称不是一个安全的协议，listener.security.protocol.map也必须设置。主机名称设置0.0.0.0绑定所有的接口，主机名称为空则绑定默认的接口。如：PLAINTEXT://myhost:9092,SSL://:9091

CLIENT://0.0.0.0:9092,REPLICATION://localhost:9093

1.4.3 broker.id

broker的唯一标识符，如果不配置则自动生成，建议配置且一定要保证集群中必须唯一，默认-1

1.4.4 log.dirs

日志数据存放的目录，如果没有配置则使用log.dir，建议此项配置。

1.4.5 message.max.bytes

服务器接受单个消息的最大大小，默认1000012 约等于976.6KB。

通过本章的介绍，相信对Kafka已经有了初步的了解，并且能够快速安装Zookeeper、Kafka组件，并且通过Kafka自身命令进行消息的发送与订阅，对服务端重要参数有了初步的认识，能够通过Java客户端编写Java程序，完成消息的发布与订阅。

第2章 生产者详解

tips 学完这一章你可以

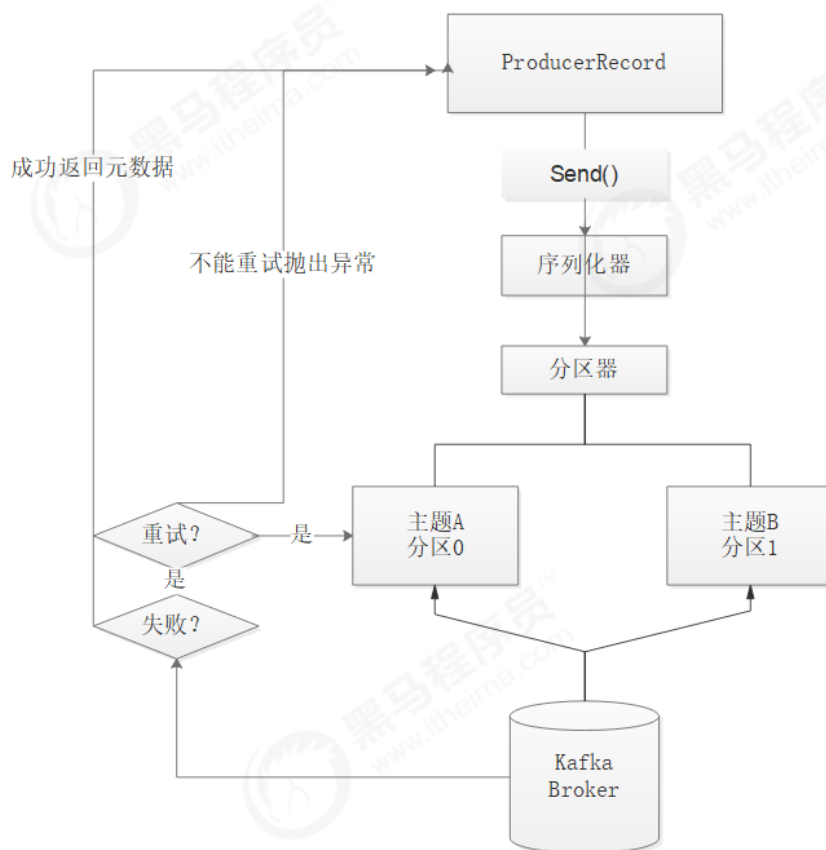
深入学习Kafka数据生产大致流程

如何创建并使用Kafka生产者

Kafka生产者常用配置

2.1 消息发送

2.1.1 Kafka Java客户端数据生产流程解析



①、首先要构造一个 ProducerRecord 对象，该对象可以声明主题Topic、分区Partition、键 Key以及值 Value，主题和值是必须要声明的，分区和键可以不用指定。

②、调用send() 方法进行消息发送。

③、因为消息要到网络上进行传输，所以必须进行序列化，序列化器的作用就是把消息的 key 和 value对象序列化成字节数组。

后，生产者就知道该往哪个主题和分区发送记录了。

⑤、接着这条记录会被添加到一个记录批次里面，这个批次里所有的消息会被发送到相同的主题和分区。会有一个独立的线程来把这些记录批次发送到相应的 Broker 上。

③、Broker成功接收到消息，表示发送成功，返回消息的元数据（包括主题和分区信息以及记录在分区里的偏移量）。发送失败，可以选择重试或者直接抛出异常。

依赖的包 <kafka.version>2.0.0</kafka.version>

```
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka_${scala.version}</artifactId>
  <version>${kafka.version}</version>
  <exclusions>
    <exclusion>
      <groupId>org.apache.zookeeper</groupId>
      <artifactId>zookeeper</artifactId>
    </exclusion>
    <exclusion>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-log4j12</artifactId>
    </exclusion>
    <exclusion>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

2.1.2 必要参数配置

见代码库：com.heima.kafka.chapter2.KafkaProducerAnalysis

```
public static Properties initConfig() {
    Properties props = new Properties();
    // 该属性指定 brokers 的地址清单，格式为 host:port。清单里不需要包含所有的 broker
    地址，
    // 生产者会从给定的 broker 里查找到其它 broker 的信息。--建议至少提供两个 broker
    的信息，因为一旦其中一个宕机，生产者仍然能够连接到集群上。
    props.put("bootstrap.servers", brokerList);
    // 将 key 转换为字节数组的配置，必须设定为一个实现了
    org.apache.kafka.common.serialization.Serializer 接口的类，
    // 生产者会用这个类把键对象序列化为字节数组。
    // --kafka 默认提供了 StringSerializer和 IntegerSerializer、
    ByteArraySerializer。当然也可以自定义序列化器。
    props.put("key.serializer",
        "org.apache.kafka.common.serialization.StringSerializer");
    // 和 key.serializer 一样，用于 value 的序列化
    props.put("value.serializer",
        "org.apache.kafka.common.serialization.StringSerializer");
}
```

```
// 内容形式如: "producer-1"
props.put("client.id", "producer.client.id.demo");
return props;
}
```

```
Properties props = initConfig();
KafkaProducer<String, String> producer = new KafkaProducer<>(props);
// KafkaProducer<String, String> producer = new KafkaProducer<>(props,
//     new StringSerializer(), new StringSerializer());
//生成 ProducerRecord 对象, 并制定 Topic, key 以及 value
ProducerRecord<String, String> record = new ProducerRecord<>(topic,
"hello, kafka!");
try {
    // 发送消息
    producer.send(record);
}
```

2.13 发送类型

发送即忘记

producer.send(record)

同步发送

```
//通过send()发送完消息后返回一个Future对象,然后调用Future对象的get()方法等待kafka响应
//如果kafka正常响应,返回一个RecordMetadata对象,该对象存储消息的偏移量
// 如果kafka发生错误,无法正常响应,就会抛出异常,我们便可以进行异常处理
producer.send(record).get();
```

异步发送

```
producer.send(record, new Callback() {
    public void onCompletion(RecordMetadata metadata, Exception exception) {
        if (exception == null) {
            System.out.println(metadata.partition() + ":" + metadata.offset());
        }
    }
});
```

2.1.4 序列化器

消息要到网络上进行传输,必须进行序列化,而序列化器的作用就是如此。

Kafka 提供了默认的字符串序列化器 (org.apache.kafka.common.serialization.StringSerializer), 还有整型 (IntegerSerializer) 和字节数组 (BytesSerializer) 序列化器,这些序列化器都实现了接口 (org.apache.kafka.common.serialization.Serializer) 基本上能够满足大部分场景的需求。

2.1.5 自定义序列化器

见代码库: com.heima.kafka.chapter2.CompanySerializer

```
/**
 * 自定义序列化器
```

北京市昌平区建材城西路金燕龙办公楼一层 电话: 400-618-9090



```
@Override
public void configure(Map configs, boolean isKey) {
}

@Override
public byte[] serialize(String topic, Company data) {
    if (data == null) {
        return null;
    }
    byte[] name, address;
    try {
        if (data.getName() != null) {
            name = data.getName().getBytes("UTF-8");
        } else {
            name = new byte[0];
        }
        if (data.getAddress() != null) {
            address = data.getAddress().getBytes("UTF-8");
        } else {
            address = new byte[0];
        }
        ByteBuffer buffer = ByteBuffer.
            allocate(4 + 4 + name.length + address.length);
        buffer.putInt(name.length);
        buffer.put(name);
        buffer.putInt(address.length);
        buffer.put(address);
        return buffer.array();
    } catch (UnsupportedEncodingException e) {
        e.printStackTrace();
    }
    return new byte[0];
}

@Override
public void close() {
}
}
```

- 使用自定义的序列化器

见代码库：com.heima.kafka.chapter2.ProducerDefineSerializer

```
public class ProducerDefineSerializer {
    public static final String brokerList = "localhost:9092";
    public static final String topic = "heima";

    public static void main(String[] args)
        throws ExecutionException, InterruptedException {
        Properties properties = new Properties();
        properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
            StringSerializer.class.getName());
        properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
            CompanySerializer.class.getName());
        // properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
        北京市昌平区建材城西路金燕龙办公楼一层 电话：400-618-9090
```



```
KafkaProducer<String, Company> producer =
    new KafkaProducer<>(properties);
Company company = Company.builder().name("kafka")
    .address("北京").build();
//    Company company = Company.builder().name("hiddenkafka")
//        .address("China").telephone("13000000000").build();
ProducerRecord<String, Company> record =
    new ProducerRecord<>(topic, company);
producer.send(record).get();
    }
}
```

2.1.6 分区器

本身kafka有自己的分区策略的，如果未指定，就会使用默认的分区策略：

Kafka根据传递消息的key来进行分区的分配，即 $\text{hash}(\text{key}) \% \text{numPartitions}$ 。如果Key相同的话，那么就会分配到统一分区。

源代码org.apache.kafka.clients.producer.internals.DefaultPartitioner分析

```
public int partition(String topic, Object key, byte[] keyBytes, Object value,
    byte[] valueBytes, Cluster cluster) {
    List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);
    int numPartitions = partitions.size();
    if (keyBytes == null) {
        int nextValue = this.nextValue(topic);
        List<PartitionInfo> availablePartitions =
            cluster.availablePartitionsForTopic(topic);
        if (availablePartitions.size() > 0) {
            int part = Utils.toPositive(nextValue) %
                availablePartitions.size();
            return
                ((PartitionInfo)availablePartitions.get(part)).partition();
        } else {
            return Utils.toPositive(nextValue) % numPartitions;
        }
    } else {
        return Utils.toPositive(Utils.murmur2(keyBytes)) % numPartitions;
    }
}
```

- 自定义分区器见代码库 com.heima.kafka.chapter2.DefinePartitioner

```
/**
 * 自定义分区器
 */
public class DefinePartitioner implements Partitioner {
    private final AtomicInteger counter = new AtomicInteger(0);

    @Override
    public int partition(String topic, Object key, byte[] keyBytes,
        北京市昌平区建材城西路金燕龙办公楼一层 电话：400-618-9090
```



```
int numPartitions = partitions.size();
if (null == keyBytes) {
    return counter.getAndIncrement() % numPartitions;
} else
    return Utils.toPositive(Utils.murmur2(keyBytes)) % numPartitions;
}
@Override
public void close() {
}
@Override
public void configure(Map<String, ?> configs) {
}
}
```

- 实现自定义分区器需要通过配置参数ProducerConfig.PARTITIONER_CLASS_CONFIG来实现

```
// 自定义分区器的使用
props.put(ProducerConfig.PARTITIONER_CLASS_CONFIG, DefinePartitioner.class.getName());
```

2.1.7 拦截器

Producer拦截器(interceptor)是个相当新的功能，它和consumer端interceptor是在Kafka 0.10版本被引入的，主要用于实现clients端的定制化控制逻辑。

生产者拦截器可以用在消息发送前做一些准备工作。

使用场景

- 1、按照某个规则过滤掉不符合要求的消息
- 2、修改消息的内容
- 3、统计类需求

见代码库：自定义拦截器com.heima.kafka.chapter2.ProducerInterceptorPrefix

```
/**
 * 自定义拦截器
 */
public class ProducerInterceptorPrefix implements
    ProducerInterceptor<String, String> {
    private volatile long sendSuccess = 0;
    private volatile long sendFailure = 0;

    @Override
    public ProducerRecord<String, String> onSend(
        ProducerRecord<String, String> record) {
        String modifiedValue = "prefix1-" + record.value();
        return new ProducerRecord<>(record.topic(),
            record.partition(), record.timestamp(),
            record.key(), modifiedValue, record.headers());
    }
    // if (record.value().length() < 5) {
    //     throw new RuntimeException();
    // }
```

```
}

@Override
public void onAcknowledgement(
    RecordMetadata recordMetadata,
    Exception e) {
    if (e == null) {
        sendSuccess++;
    } else {
        sendFailure++;
    }
}

@Override
public void close() {
    double successRatio = (double) sendSuccess / (sendFailure +
sendSuccess);
    System.out.println("[INFO] 发送成功率="
        + String.format("%f", successRatio * 100) + "%");
}

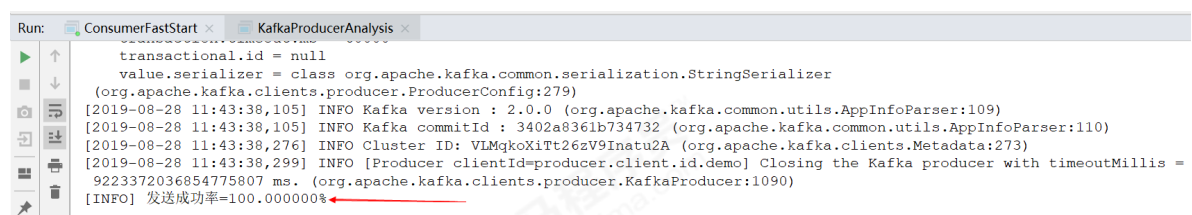
@Override
public void configure(Map<String, ?> map) {
}
}
```

- 实现自定义拦截器之后需要在配置参数中指定这个拦截器，此参数的默认值为空，如下：

```
// 自定义拦截器使用
props.put(ProducerConfig.INTERCEPTOR_CLASSES_CONFIG, ProducerDefineSerializer.class.getName());
```

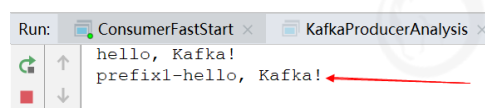
- 功能演示：

发送端



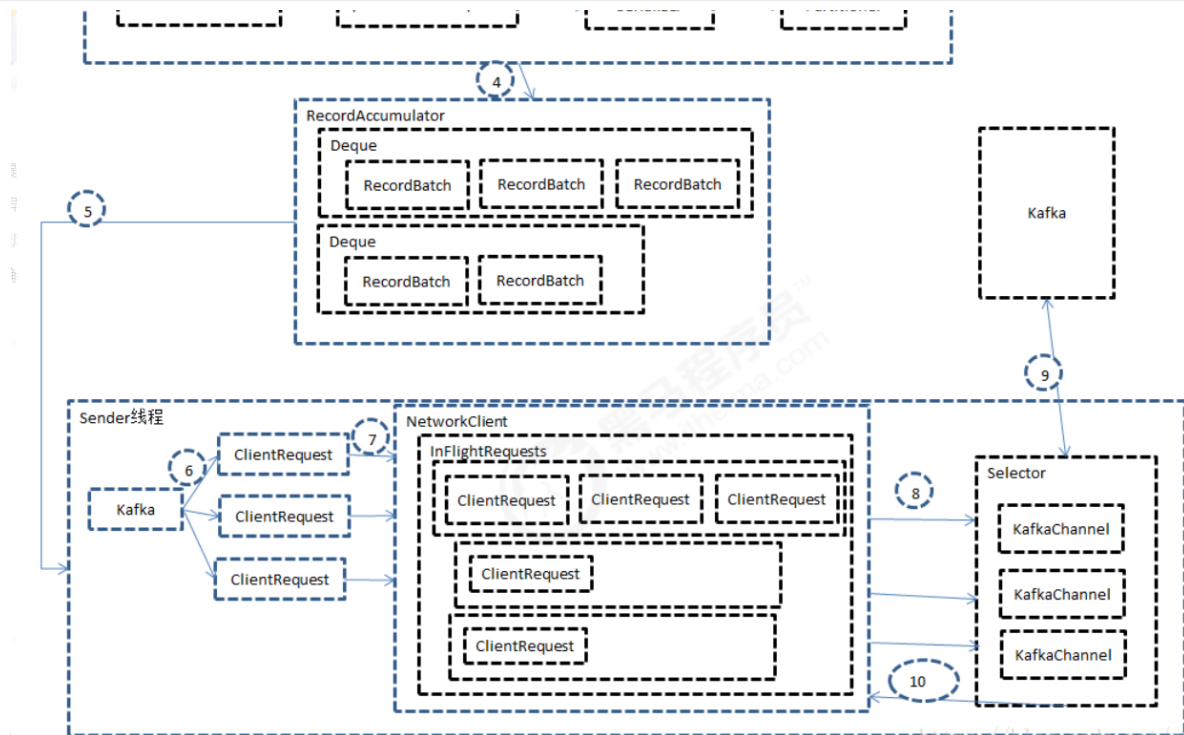
The screenshot shows the console output of the KafkaProducerAnalysis application. It displays the configuration of the Kafka producer, including the transactional ID, serializer, and commit ID. The output shows that the producer is successfully sending messages to the Kafka cluster. The final line of the output is "[INFO] 发送成功率=100.000000%", indicating that all messages were sent successfully.

接收端



The screenshot shows the console output of the ConsumerFastStart application. It displays the messages received from the Kafka cluster. The first message is "hello, Kafka!" and the second message is "prefix1-hello, Kafka!". The output is shown in a console window with a red arrow pointing to the second message.

2.2 发送原理剖析



消息发送的过程中，涉及到两个线程协同工作，主线程首先将业务数据封装成ProducerRecord对象，之后调用send()方法将消息放入RecordAccumulator(消息收集器，也可以理解为主线程与Sender线程直接的缓冲区)中暂存，Sender线程负责将消息信息构成请求，并最终执行网络I/O的线程，它从RecordAccumulator中取出消息并批量发送出去，需要注意的是，KafkaProducer是线程安全的，多个线程间可以共享使用同一个KafkaProducer对象

2.3 其他生产者参数

之前提及的默认三个客户端参数，大部分参数都有合理的默认值，一般情况下不需要修改它们，

参考官网：<http://kafka.apache.org/documentation/#producerconfigs>

2.3.1 acks

这个参数用来指定分区中必须有多少个副本收到这条消息，之后生产者才会认为这条消息时写入成功的。acks是生产者客户端中非常重要的一个参数，它涉及到消息的可靠性和吞吐量之间的权衡。

- ack=0，生产者在成功写入消息之前不会等待任何来自服务器的相应。如果出现问题生产者是感知不到的，消息就丢失了。不过因为生产者不需要等待服务器响应，所以它可以以网络能够支持的最大速度发送消息，从而达到很高的吞吐量。
- ack=1，默认值为1，只要集群的首领节点收到消息，生产这就会收到一个来自服务器的成功响应。如果消息无法达到首领节点（比如首领节点崩溃，新的首领还没有被选举出来），生产者会收到一个错误响应，为了避免数据丢失，生产者会重发消息。但是，这样还有可能会导致数据丢失，如果收到写成功通知，此时首领节点还没来的及同步数据到follower节点，首领节点崩溃，就会导致数据丢失。
- ack=-1，只有当所有参与复制的节点都收到消息时，生产这会收到一个来自服务器的成功响应，这种模式是最安全的，它可以保证不止一个服务器收到消息。

注意：acks参数配置的是一个字符串类型，而不是整数类型，如果配置为整数类型会抛出以下异常



```
at org.apache.kafka.common.config.ConfigDef.parseType(ConfigDef.java:668)
at org.apache.kafka.common.config.ConfigDef.parseValue(ConfigDef.java:469)
at org.apache.kafka.common.config.ConfigDef.parse(ConfigDef.java:462)
at org.apache.kafka.common.config.AbstractConfig.<init>(AbstractConfig.java:62)
at org.apache.kafka.common.config.AbstractConfig.<init>(AbstractConfig.java:75)
at org.apache.kafka.clients.producer.ProducerConfig.<init>(ProducerConfig.java:364)
at org.apache.kafka.clients.producer.KafkaProducer.<init>(KafkaProducer.java:304)
at com.heimakafka.chapter2.KafkaProducerAnalysis.main(KafkaProducerAnalysis.java:65)
```

2.3.2 retries

生产者从服务器收到的错误有可能是临时性的错误（比如分区找不到首领）。在这种情况下，如果达到了 `retries` 设置的次数，生产者会放弃重试并返回错误。默认情况下，生产者会在每次重试之间等待 100ms，可以通过 `retry.backoff.ms` 参数来修改这个时间间隔。

2.3.3 batch.size

当有多个消息要被发送到同一个分区时，生产者会把它们放在同一个批次里。该参数指定了一个批次可以使用的内存大小，按照字节数计算，而不是消息个数。当批次被填满，批次里的所有消息会被发送出去。不过生产者并不一定会等到批次被填满才发送，半满的批次，甚至只包含一个消息的批次也可能被发送。所以就算把 `batch.size` 设置的很大，也不会造成延迟，只会占用更多的内存而已，如果设置的太小，生产者会因为频繁发送消息而增加一些额外的开销。

2.3.4 max.request.size

该参数用于控制生产者发送的请求大小，它可以指定能发送的单个消息的最大值，也可以指单个请求里所有消息的总大小。`broker` 对可接收的消息最大值也有自己的限制（`message.max.size`），所以两边的配置最好匹配，避免生产者发送的消息被 `broker` 拒绝。

总结

本章主要讲了生产者客户端的用法以及整体流程架构，主要内容包括配置参数的详解、消息的发送方式、序列化器、分区器、拦截器等，在实际使用中，Kafka已经提供了良好的Java客户端支持，提高了开发效率。

第3章 消费者详解

tips 学完这一章你可以

深入学习Kafka数据消费大致流程

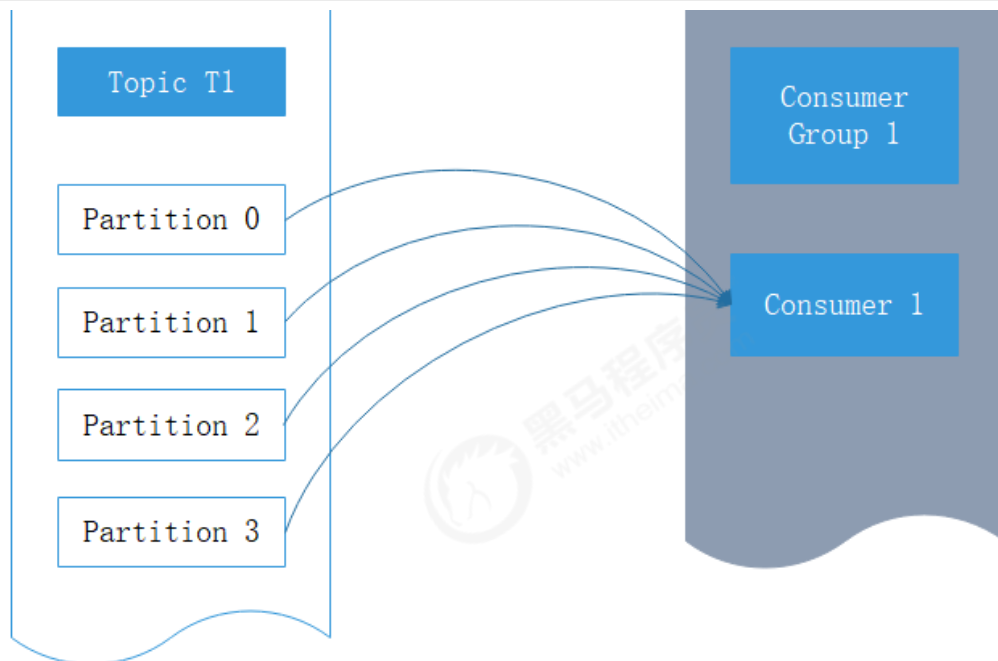
如何创建并使用Kafka消费者

Kafka消费者常用配置

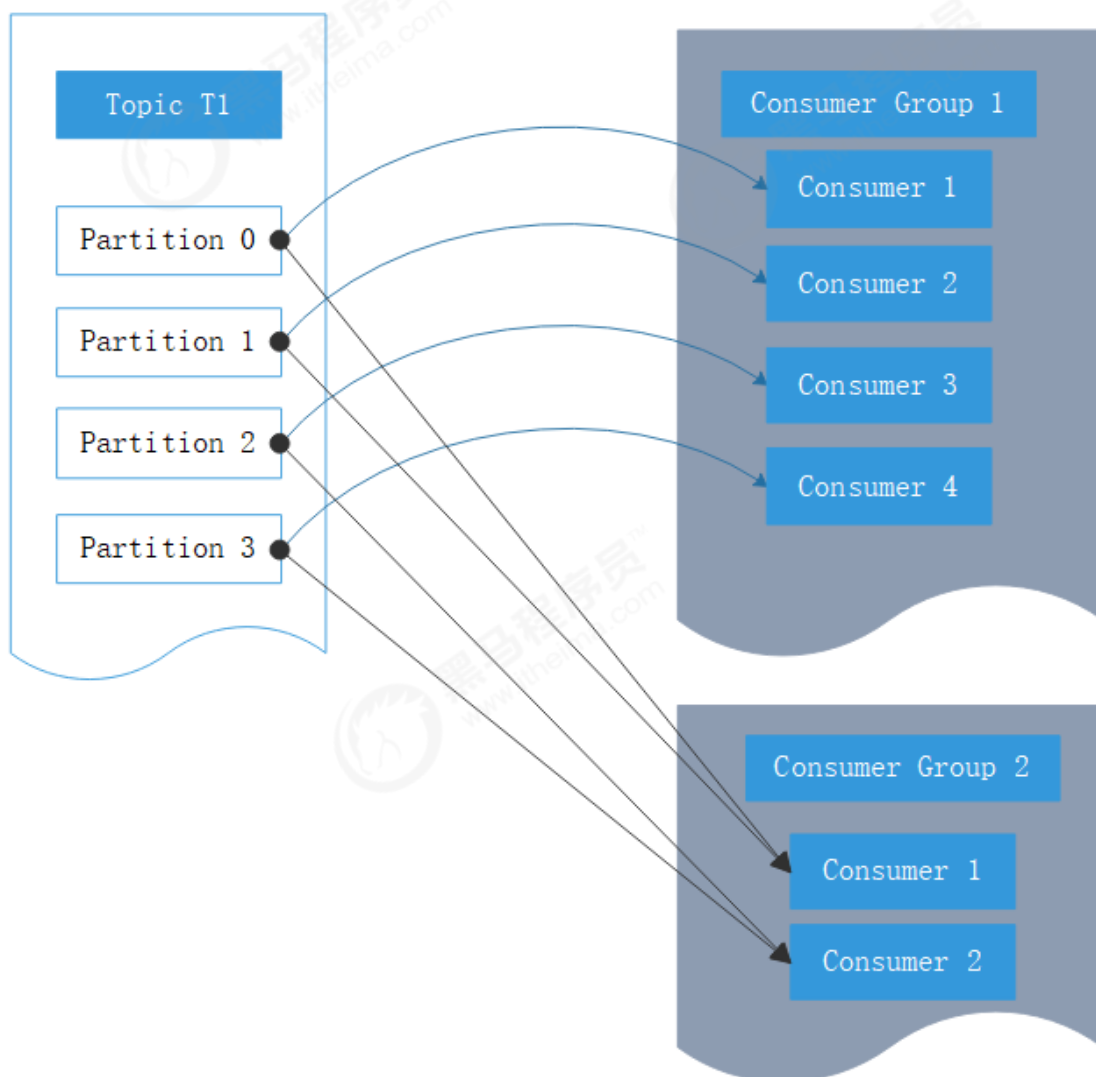
3.1 概念入门

3.1.1 消费者和消费组

Kafka消费者是**消费组**的一部分，当多个消费者形成一个消费组来消费主题时，每个消费者会收到不同分区的消息。假设有一个T1主题，该主题有4个分区；同时我们有一个消费组G1，这个消费组只有一个消费者C1。那么消费者C1将会收到这4个分区的消息，如下所示：



Kafka一个很重要的特性就是，只需写入一次消息，可以支持任意多的应用读取这个消息。换句话说，每个应用都可以读到全量的消息。为了使得每个应用都能读到全量消息，应用需要有不同的消费组。对于上面的例子，假如我们新增了一个新的消费组G2，而这个消费组有两个消费者，那么会是这样的：



3.2 消息接收

见代码库：com.heima.kafka.chapter3.KafkaConsumerAnalysis

3.2.1 必要参数设置

KafkaConsumer实例中参数众多，后续会深入讲解

```
public static Properties initConfig() {
    Properties props = new Properties();
    // 与KafkaProducer中设置保持一致
    props.put("key.deserializer",
        "org.apache.kafka.common.serialization.StringDeserializer");
    props.put("value.deserializer",
        "org.apache.kafka.common.serialization.StringDeserializer");
    // 必填参数，该参数和KafkaProducer中的相同，制定连接Kafka集群所需的broker地址清单，可以设置一个或者多个
    props.put("bootstrap.servers", brokerList);
    // 消费者隶属于的消费组，默认为空，如果设置为空，则会抛出异常，这个参数要设置成具有一定业务含义的名称
    props.put("group.id", groupId);
    // 指定KafkaConsumer对应的客户端ID，默认为空，如果不设置KafkaConsumer会自动生成一个非空字符串
    props.put("client.id", "consumer.client.id.demo");
    return props;
}
```

3.2.2 订阅主题和分区

创建完消费者后我们便可以订阅主题了，只需要通过调用subscribe()方法即可，这个方法接收一个主题列表

```
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Arrays.asList(topic));
```

另外，我们也可以使用正则表达式来匹配多个主题，而且订阅之后如果又有匹配的新主题，那么这个消费组会立即对其进行消费。正则表达式在连接Kafka与其他系统时非常有用。比如订阅所有的测试主题：

```
consumer.subscribe(Pattern.compile("heima*"));
```

指定订阅的分区

```
// 指定订阅的分区
consumer.assign(Arrays.asList(new TopicPartition("topic0701", 0)));
```

3.2.2 反序列化

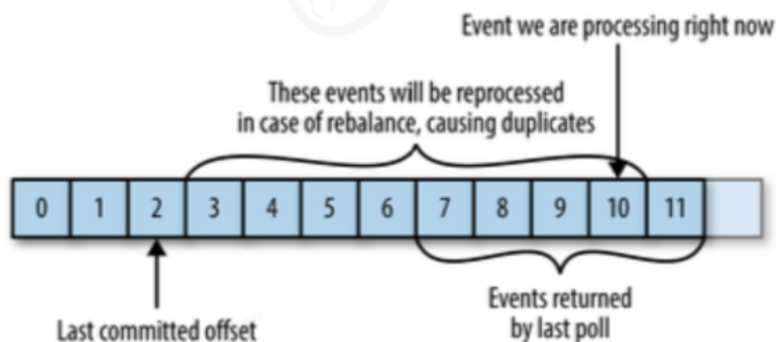
```
props.put("key.serializer",  
"org.apache.kafka.common.serialization.StringSerializer");  
props.put("value.deserializer",  
"org.apache.kafka.common.serialization.StringDeserializer");
```

3.2.3 位移提交

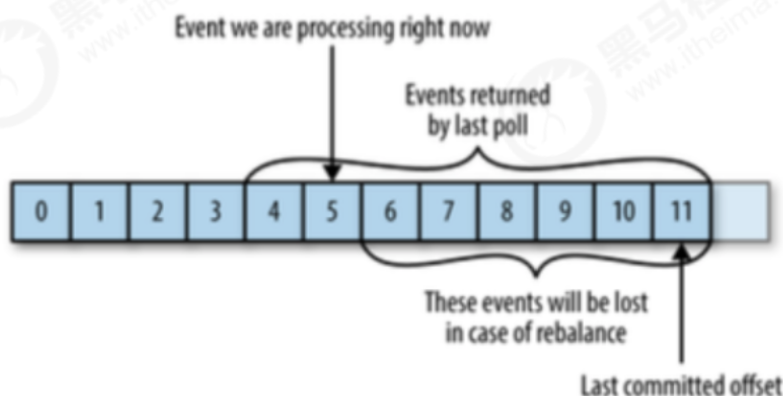
对于Kafka中的分区而言，它的每条消息都有唯一的offset，用来表示消息在分区中的位置。

当我们调用poll()时，该方法会返回我们没有消费的消息。当消息从broker返回消费者时，broker并不跟踪这些消息是否被消费者接收到；Kafka让消费者自身来管理消费的位移，并向消费者提供更新位移的接口，这种更新位移方式称为提交（commit）。

重复消费



消息丢失



自动提交

这种方式让消费者来管理位移，应用本身不需要显式操作。当我们将enable.auto.commit设置为true，那么消费者会在poll方法调用后每隔5秒（由auto.commit.interval.ms指定）提交一次位移。和很多其他操作一样，自动提交也是由poll()方法来驱动的；在调用poll()时，消费者判断是否到达提交时间，如果是则提交上一次poll返回的最大位移。

需要注意到，这种方式可能会导致消息重复消费。假如，某个消费者poll消息后，应用正在处理消息，在3秒后Kafka进行了重平衡，那么由于没有更新位移导致重平衡后这部分消息重复消费。

同步提交

见代码库：com.heima.kafka.chapter3.CheckOffsetAndCommit



```

        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
            StringDeserializer.class.getName());
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
            StringDeserializer.class.getName());
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, brokerList);
        props.put(ConsumerConfig.GROUP_ID_CONFIG, groupId);
        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");

        // 手动提交开启
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);
        return props;
    }

```

```

while (true) {
    ConsumerRecords<String, String> records = consumer.poll(1000);
    if (records.isEmpty()) {
        break;
    }
    List<ConsumerRecord<String, String>> partitionRecords =
records.records(tp);
    lastConsumedOffset = partitionRecords.get(partitionRecords.size() -
1).offset();
    consumer.commitSync(); //同步提交消费位移
}

```

异步提交

手动提交有一个缺点，那就是当发起提交调用时应用会阻塞。当然我们可以减少手动提交的频率，但这个会增加消息重复的概率（和自动提交一样）。另外一个解决办法是，使用异步提交的API。

见代码：com.heima.kafka.chapter3.OffsetCommitAsyncCallback

但是异步提交也有个缺点，那就是如果服务器返回提交失败，异步提交不会进行重试。相比较起来，同步提交会进行重试直到成功或者最后抛出异常给应用。异步提交没有实现重试是因为，如果同时存在多个异步提交，进行重试可能会导致位移覆盖。举个例子，假如我们发起了一个异步提交commitA，此时的提交位移为2000，随后又发起了一个异步提交commitB且位移为3000；commitA提交失败但commitB提交成功，此时commitA进行重试并成功的话，会将实际上已经提交的位移从3000回滚到2000，导致消息重复消费。

异步回调

```

try {
    while (running.get()) {
        ConsumerRecords<String, String> records = consumer.poll(1000);
        for (ConsumerRecord<String, String> record : records) {
            //do some logical processing.
        }
        // 异步回调
        consumer.commitAsync(new OffsetCommitCallback() {
            @Override
            public void onComplete(Map<TopicPartition,
OffsetAndMetadata> offsets,
                                Exception exception) {

```



```
        } else {  
            log.error("fail to commit offsets {}", offsets,  
exception);  
        }  
    }  
});  
}  
} finally {  
    consumer.close();  
}
```

3.2.4 指定位移消费

到目前为止，我们知道消息的拉取是根据poll()方法中的逻辑来处理的，但是这个方法对于普通开发人员来说就是个黑盒处理，无法精确掌握其消费的起始位置。

seek()方法正好提供了这个功能，让我们得以追踪以前的消费或者回溯消费。

见代码库：com.heima.kafka.chapter3.SeekDemo

```
/**  
 * 指定位移消费  
 */  
public class SeekDemo extends ConsumerClientConfig {  
  
    public static void main(String[] args) {  
        Properties props = initConfig();  
        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);  
        consumer.subscribe(Arrays.asList(topic));  
        // timeout参数设置多少合适？太短会使分区分配失败，太长又有可能造成一些不必要的等待  
        consumer.poll(Duration.ofMillis(2000));  
        // 获取消费者所分配到的分区  
        Set<TopicPartition> assignment = consumer.assignment();  
        System.out.println(assignment);  
        for (TopicPartition tp : assignment) {  
            // 参数partition表示分区，offset表示指定从分区的哪个位置开始消费  
            consumer.seek(tp, 10);  
        }  
        // consumer.seek(new TopicPartition(topic,0),10);  
        while (true) {  
            ConsumerRecords<String, String> records =  
consumer.poll(Duration.ofMillis(1000));  
            //consume the record.  
            for (ConsumerRecord<String, String> record : records) {  
                System.out.println(record.offset() + ":" + record.value());  
            }  
        }  
    }  
}
```

增加判断是否分配到了分区，见代码库：com.heima.kafka.chapter3.SeekDemoAssignment

```
public static void main(String[] args) {  
    Properties props = initConfig();  
    北京市昌平区建材城西路金燕龙办公楼一层 电话：400-618-9090
```



```

        long start = System.currentTimeMillis();
        Set<TopicPartition> assignment = new HashSet<>();
        while (assignment.size() == 0) {
            consumer.poll(Duration.ofMillis(100));
            assignment = consumer.assignment();
        }
        long end = System.currentTimeMillis();
        System.out.println(end - start);
        System.out.println(assignment);
        for (TopicPartition tp : assignment) {
            consumer.seek(tp, 10);
        }
        while (true) {
            ConsumerRecords<String, String> records =
consumer.poll(Duration.ofMillis(1000));
            //consume the record.
            for (ConsumerRecord<String, String> record : records) {
                System.out.println(record.offset() + ":" + record.value());
            }
        }
    }
}

```

- 指定从分区末尾开始消费，见代码库：com.heima.kafka.chapter3.SeekToEnd

```

// 指定从分区末尾开始消费
Map<TopicPartition, Long> offsets = consumer.endOffsets(assignment);
for (TopicPartition tp : assignment) {
    consumer.seek(tp, offsets.get(tp));
}

```

- 演示位移越界操作，修改代码如下：

```

for (TopicPartition tp : assignment) {
    //consumer.seek(tp, offsets.get(tp));
    consumer.seek(tp, offsets.get(tp) + 1);
}

```

会通过auto.offset.reset参数的默认值将位置重置，效果如下：

```

INFO [Consumer clientId=consumer-1, groupId=group.heima] Fetch offset 1 is out
of range for partition heima-0, resetting offset
(org.apache.kafka.clients.consumer.internals.Fetcher:967)
INFO [Consumer clientId=consumer-1, groupId=group.heima] Fetch offset 10 is out
of range for partition heima-1, resetting offset
(org.apache.kafka.clients.consumer.internals.Fetcher:967)
INFO [Consumer clientId=consumer-1, groupId=group.heima] Resetting offset for
partition heima-0 to offset 0.
(org.apache.kafka.clients.consumer.internals.Fetcher:583)
INFO [Consumer clientId=consumer-1, groupId=group.heima] Resetting offset for
partition heima-1 to offset 9.
(org.apache.kafka.clients.consumer.internals.Fetcher:583)

```




再均衡是指分区的所属从一个消费者转移到另外一个消费者的行为，它为消费组具备了高可用性和伸缩性提供了保障，使得我们既方便又安全地删除消费组内的消费者或者往消费组内添加消费者。不过再均衡发生期间，消费者是无法拉取消息的。

见代码库：com.heima.kafka.chapter3.CommitSyncInRebalance

```
public static void main(String[] args) {
    Properties props = initConfig();
    KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);

    Map<TopicPartition, OffsetAndMetadata> currentOffsets = new HashMap<>();
    consumer.subscribe(Arrays.asList(topic), new ConsumerRebalanceListener()
    {
        @Override
        public void onPartitionsRevoked(Collection<TopicPartition>
partitions) {
            // 尽量避免重复消费
            consumer.commitSync(currentOffsets);
        }

        @Override
        public void onPartitionsAssigned(Collection<TopicPartition>
partitions) {
            //do nothing.
        }
    });

    try {
        while (isRunning.get()) {
            ConsumerRecords<String, String> records =
                consumer.poll(Duration.ofMillis(1000));
            for (ConsumerRecord<String, String> record : records) {
                System.out.println(record.offset() + ":" + record.value());
                // 异步提交消费位移，在发生再均衡动作之前可以通过再均衡监听器的
                onPartitionsRevoked回调执行commitSync方法同步提交位移。
                currentOffsets.put(new TopicPartition(record.topic(),
record.partition()),
                    new OffsetAndMetadata(record.offset() + 1));
            }
            consumer.commitAsync(currentOffsets, null);
        }
    } finally {
        consumer.close();
    }
}
```

3.2.5 消费者拦截器

之前章节讲了生产者拦截器，对应的消费者也有相应的拦截器概念，消费者拦截器主要是在消费到消息或者在提交消费位移时进行的一些定制化的操作。

使用场景

见代码库：com.heima.kafka.chapter3.ConsumerInterceptorTTL

```
public ConsumerRecords<String, String> onConsume(ConsumerRecords<String, String>
records) {
    System.out.println("before:" + records);
    long now = System.currentTimeMillis();
    Map<TopicPartition, List<ConsumerRecord<String, String>>> newRecords
        = new HashMap<>();
    for (TopicPartition tp : records.partitions()) {
        List<ConsumerRecord<String, String>> tpRecords =
records.records(tp);
        List<ConsumerRecord<String, String>> newTpRecords = new ArrayList<>
();
        for (ConsumerRecord<String, String> record : tpRecords) {
            if (now - record.timestamp() < EXPIRE_INTERVAL) {
                newTpRecords.add(record);
            }
        }
        if (!newTpRecords.isEmpty()) {
            newRecords.put(tp, newTpRecords);
        }
    }
    return new ConsumerRecords<>(newRecords);
}
```

实现自定义拦截器之后，需要在KafkaConsumer中配置指定这个拦截器，如下

```
// 指定消费者拦截器
props.put(ConsumerConfig.INTERCEPTOR_CLASSES_CONFIG, ConsumerInterceptorTTL.class
.getName());
```

效果演示

发送端同时发送两条消息，其中一条修改timestamp的值来使其变得超时，如下：

com.heima.kafka.chapter3.ProducerFastStart

```
ProducerRecord<String, String> record = new ProducerRecord<>(topic, "Kafka-demo-
001", "hello, kafka!");
ProducerRecord<String, String> record2 = new ProducerRecord<>(topic, 0,
System.currentTimeMillis() - 10 * 1000, "Kafka-demo-001", "hello, kafka!->超时");
```

启动消费端运行如下，只收到了未超时的消息：

```
before:org.apache.kafka.clients.consumer.ConsumerRecords@7bd7d6d6
topic = heima, partition = 1, offset = 15
key = Kafka-demo-001, value = hello, Kafka!
```

3.2.6 消费者参数补充

fetch.min.bytes

说，这个参数可以减少broker和消费者的压力，因为减少了往返的时间。而对于有大量消费者的主题来说，则可以明显减轻broker压力。

fetch.max.wait.ms

上面的fetch.min.bytes参数指定了消费者读取的最小数据量，而这个参数则指定了消费者读取时最长等待时间，从而避免长时间阻塞。这个参数默认为500ms。

max.partition.fetch.bytes

这个参数指定了每个分区返回的最多字节数，默认为1M。也就是说，KafkaConsumer.poll()返回记录列表时，每个分区的记录字节数最多为1M。如果一个主题有20个分区，同时有5个消费者，那么每个消费者需要4M的空间来处理消息。实际情况中，我们需要设置更多的空间，这样当存在消费者宕机时，其他消费者可以承担更多的分区。

max.poll.records

这个参数控制一个poll()调用返回的记录数，这个可以用来控制应用在拉取循环中的处理数据量。

更多参数见官网: <http://kafka.apache.org/documentation/#consumerconfigs>

总结

本章主要讲解了消费者和消费组的概念，以及如何正确的使用KafkaConsumer，其中重点讲解了参数的配置，订阅、反序列化、位移提交、再均衡、拦截器等知识点。

第4章 主题

tips 学完这一章你可以

深入学习Kafka主题的管理

KafkaAdminClient应用

4.1 管理

4.1.1 创建主题

命令

```
bin/kafka-topics.sh --zookeeper localhost:2181 --create --topic heima --partitions 2 --replication-factor 1
```

localhost:2181 zookeeper所在的ip，zookeeper 必传参数，多个zookeeper用‘,’分开。

partitions 用于设置主题分区数，每个线程处理一个分区数据

replication-factor 用于设置主题副本数，每个副本分布在不通节点，不能超过总结点数。如你只有一个节点，但是创建时指定副本数为2，就会报错。

查看topic元数据信息的方法

topic元数据信息保存在Zookeeper节点中



```
itcast@Server-node:/mnt/d/zookeeper-3.4.14$ bin/zkCli.sh -server localhost:2181
Connecting to localhost:2181
.....
[zk: localhost:2181(CONNECTED) 2] get /brokers/topics/heima
{"version":1,"partitions":{"1":[0],"0":[0]}}
cZxid = 0x618
ctime = Wed Aug 28 05:51:35 GMT 2019
mZxid = 0x618
mtime = Wed Aug 28 05:51:35 GMT 2019
pZxid = 0x619
cversion = 1
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 44
numChildren = 1
[zk: localhost:2181(CONNECTED) 3]
```

4.1.2 查看主题

```
// 查看所有主题
itcast@Server-node:/mnt/d/kafka_2.12-2.2.1$ bin/kafka-topics.sh --list --
zookeeper localhost:2181
__consumer_offsets
__transaction_state
_schemas
heima
topic0701
topic0703
topic0703kafka_source
topic0828
itcast@Server-node:/mnt/d/kafka_2.12-2.2.1$

// 查看某个特定主题信息,不指定topic则查询所有 通过 --describe
itcast@Server-node:/mnt/d/kafka_2.12-2.2.1$ bin/kafka-topics.sh --describe --
zookeeper localhost:2181 --topic heima
Topic:heima      PartitionCount:2      ReplicationFactor:1      Configs:
Topic: heima     Partition: 0          Leader: 0                 Replicas: 0             Isr: 0
Topic: heima     Partition: 1          Leader: 0                 Replicas: 0             Isr: 0
itcast@Server-node:/mnt/d/kafka_2.12-2.2.1$
//查看正在同步的主题
// 通过 --describe 和 under-replicated-partitions命令组合查看 under-replacation状态
```

4.1.3 修改主题

```
// 增加配置
itcast@Server-node:/mnt/d/kafka_2.12-2.2.1$ bin/kafka-topics.sh --alter --
zookeeper localhost:2181 --topic heima
--config flush.messages=1
WARNING: Altering topic configuration from this script has been deprecated and
may be removed in future releases.
        Going forward, please use kafka-configs.sh for this functionality
Updated config for topic heima
```



```
itcast@Server-node:/mnt/d/kafka_2.12-2.2.1$ bin/kafka-topics.sh --alter --
zookeeper localhost:2181 --topic heima --delete-config flush.messages
WARNING: Altering topic configuration from this script has been deprecated and
may be removed in future releases.
    Going forward, please use kafka-configs.sh for this functionality
Updated config for topic heima.
```

4.1.4 删除主题

若 delete.topic.enable=true 直接彻底删除该 Topic。若 delete.topic.enable=false 如果当前 Topic 没有使用过即没有传输过信息：可以彻底删除。如果当前 Topic 有使用过即有过传输过信息：并没有真正删除 Topic 只是把这个 Topic 标记为删除(marked for deletion)，重启 Kafka Server 后删除。

```
itcast@Server-node:/mnt/d/kafka_2.12-2.2.1$ bin/kafka-topics.sh --delete --
zookeeper localhost:2181 --topic heima
Topic heima is marked for deletion.
Note: This will have no impact if delete.topic.enable is not set to true.

// 标记为 marked for deletion
itcast@Server-node:/mnt/d/kafka_2.12-2.2.1$ bin/kafka-topics.sh --list --
zookeeper localhost:2181
__consumer_offsets
topic0701
heima - marked for deletion
```

4.2 增加分区

```
// 增加分区数
itcast@Server-node:/mnt/d/kafka_2.12-2.2.1$ bin/kafka-topics.sh --alter --
zookeeper localhost:2181 --topic heima --partitions 3
WARNING: If partitions are increased for a topic that has a key, the partition
logic or ordering of the messages will be affected
Adding partitions succeeded!

//修改分区数时，仅能增加分区个数。若是用其减少 partition 个数，则会报如下错误信息：
itcast@Server-node:/mnt/d/kafka_2.12-2.2.1$ bin/kafka-topics.sh --alter --
zookeeper localhost:2181 --topic heima --partitions 2
WARNING: If partitions are increased for a topic that has a key, the partition
logic or ordering of the messages will be affected
Error while executing topic command : The number of partitions for a topic can
only be increased. Topic heima currently has 3 partitions, 2 would not be an
increase.
[2019-08-28 08:43:41,478] ERROR
org.apache.kafka.common.errors.InvalidPartitionsException: The number of
partitions for a topic can only be increased. Topic heima currently has 3
partitions, 2 would not be an increase.
(kafka.admin.TopicCommand$)
```

4.3 分区副本的分配—只做了解



见官方文档：<http://kafka.apache.org/documentation/#topicconfigs>

Configurations pertinent to topics have both a server default as well an optional per-topic override. If no per-topic configuration is given the server default is used. The override can be set at topic creation time by giving one or more --config options. This example creates a topic named my-topic with a custom max message size and flush rate:

```
1
2
> bin/kafka-topics.sh --bootstrap-server localhost:9092 --create --topic my-
topic --partitions 1 \
    --replication-factor 1 --config max.message.bytes=64000 --config
flush.messages=1
Overrides can also be changed or set later using the alter configs command. This
example updates the max message size for my-topic:
```

```
1
2
> bin/kafka-configs.sh --zookeeper localhost:2181 --entity-type topics --entity-
name my-topic
    --alter --add-config max.message.bytes=128000
To check overrides set on the topic you can do
```

```
1
2
> bin/kafka-configs.sh --zookeeper localhost:2181 --entity-type topics --entity-
name my-topic --describe
To remove an override you can do
```

```
1
2
> bin/kafka-configs.sh --zookeeper localhost:2181 --entity-type topics --
entity-name my-topic
    --alter --delete-config max.message.bytes
```

The following are the topic-level configurations. The server's default configuration for this property is given under the Server Default Property heading. A given server default config value only applies to a topic if it does not have an explicit topic config override.

4.5 KafkaAdminClient应用

我们都习惯使用Kafka中bin目录下的脚本工具来管理查看Kafka，但是有些时候需要将某些管理查看的功能集成到系统（比如Kafka Manager）中，那么就需要调用一些API来直接操作Kafka了。

见代码库：`com.heima.kafka.chapter4.KafkaAdminConfigOperation`

```
/**
 * kafkaAdminClient应用
 */
public class KafkaAdminConfigOperation {

    public static void main(String[] args) throws ExecutionException,
InterruptedException {
        // describeTopicConfig();
        // alterTopicConfig();
        addTopicPartitions();
    }
}
```




```
//Config(entries=[ConfigEntry(name=compression.type, value=producer,
source=DEFAULT_CONFIG, isSensitive=false, isReadOnly=false, synonyms=[]),
ConfigEntry(name=leader.replication.throttled.replicas, value=,
source=DEFAULT_CONFIG, isSensitive=false, isReadOnly=false, synonyms=[]),
ConfigEntry(name=message.downconversion.enable, value=true,
source=DEFAULT_CONFIG, isSensitive=false, isReadOnly=false, synonyms=[]),
ConfigEntry(name=min.insync.replicas, value=1, source=DEFAULT_CONFIG,
isSensitive=false, isReadOnly=false, synonyms=[]),
ConfigEntry(name=segment.jitter.ms, value=0, source=DEFAULT_CONFIG,
isSensitive=false, isReadOnly=false, synonyms=[]),
ConfigEntry(name=cleanup.policy, value=delete, source=DEFAULT_CONFIG,
isSensitive=false, isReadOnly=false, synonyms=[]), ConfigEntry(name=flush.ms,
value=9223372036854775807, source=DEFAULT_CONFIG, isSensitive=false,
isReadOnly=false, synonyms=[]),
ConfigEntry(name=follower.replication.throttled.replicas, value=,
source=DEFAULT_CONFIG, isSensitive=false, isReadOnly=false, synonyms=[]),
ConfigEntry(name=segment.bytes, value=1073741824, source=STATIC_BROKER_CONFIG,
isSensitive=false, isReadOnly=false, synonyms=[]),
ConfigEntry(name=retention.ms, value=604800000, source=DEFAULT_CONFIG,
isSensitive=false, isReadOnly=false, synonyms=[]),
ConfigEntry(name=flush.messages, value=9223372036854775807,
source=DEFAULT_CONFIG, isSensitive=false, isReadOnly=false, synonyms=[]),
ConfigEntry(name=message.format.version, value=2.0-IV1, source=DEFAULT_CONFIG,
isSensitive=false, isReadOnly=false, synonyms=[]),
ConfigEntry(name=file.delete.delay.ms, value=60000, source=DEFAULT_CONFIG,
isSensitive=false, isReadOnly=false, synonyms=[]),
ConfigEntry(name=max.message.bytes, value=1000012, source=DEFAULT_CONFIG,
isSensitive=false, isReadOnly=false, synonyms=[]),
ConfigEntry(name=min.compaction.lag.ms, value=0, source=DEFAULT_CONFIG,
isSensitive=false, isReadOnly=false, synonyms=[]),
ConfigEntry(name=message.timestamp.type, value=CreateTime,
source=DEFAULT_CONFIG, isSensitive=false, isReadOnly=false, synonyms=[]),
ConfigEntry(name=preallocate, value=false, source=DEFAULT_CONFIG,
isSensitive=false, isReadOnly=false, synonyms=[]),
ConfigEntry(name=min.cleanable.dirty.ratio, value=0.5, source=DEFAULT_CONFIG,
isSensitive=false, isReadOnly=false, synonyms=[]),
ConfigEntry(name=index.interval.bytes, value=4096, source=DEFAULT_CONFIG,
isSensitive=false, isReadOnly=false, synonyms=[]),
ConfigEntry(name=unclean.leader.election.enable, value=false,
source=DEFAULT_CONFIG, isSensitive=false, isReadOnly=false, synonyms=[]),
ConfigEntry(name=retention.bytes, value=-1, source=DEFAULT_CONFIG,
isSensitive=false, isReadOnly=false, synonyms=[]),
ConfigEntry(name=delete.retention.ms, value=86400000, source=DEFAULT_CONFIG,
isSensitive=false, isReadOnly=false, synonyms=[]), ConfigEntry(name=segment.ms,
value=604800000, source=DEFAULT_CONFIG, isSensitive=false, isReadOnly=false,
synonyms=[]), ConfigEntry(name=message.timestamp.difference.max.ms,
value=9223372036854775807, source=DEFAULT_CONFIG, isSensitive=false,
isReadOnly=false, synonyms=[]), ConfigEntry(name=segment.index.bytes,
value=10485760, source=DEFAULT_CONFIG, isSensitive=false, isReadOnly=false,
synonyms=[]))])

public static void describeTopicConfig() throws ExecutionException,
    InterruptedException {
    String brokerList = "localhost:9092";
    String topic = "heima";

    Properties props = new Properties();
```



```
AdminClient client = AdminClient.create(props);

ConfigResource resource =
    new ConfigResource(ConfigResource.Type.TOPIC, topic);
DescribeConfigsResult result =
    client.describeConfigs(Collections.singleton(resource));
Config config = result.all().get().get(resource);
System.out.println(config);
client.close();
}

public static void alterTopicConfig() throws ExecutionException,
InterruptedException {
    String brokerList = "localhost:9092";
    String topic = "heima";

    Properties props = new Properties();
    props.put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG, brokerList);
    props.put(AdminClientConfig.REQUEST_TIMEOUT_MS_CONFIG, 30000);
    AdminClient client = AdminClient.create(props);

    ConfigResource resource =
        new ConfigResource(ConfigResource.Type.TOPIC, topic);
    ConfigEntry entry = new ConfigEntry("cleanup.policy", "compact");
    Config config = new Config(Collections.singleton(entry));
    Map<ConfigResource, Config> configs = new HashMap<>();
    configs.put(resource, config);
    AlterConfigsResult result = client.alterConfigs(configs);
    result.all().get();

    client.close();
}

public static void addTopicPartitions() throws ExecutionException,
InterruptedException {
    String brokerList = "localhost:9092";
    String topic = "heima";

    Properties props = new Properties();
    props.put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG, brokerList);
    props.put(AdminClientConfig.REQUEST_TIMEOUT_MS_CONFIG, 30000);
    AdminClient client = AdminClient.create(props);

    NewPartitions newPartitions = NewPartitions.increaseTo(5);
    Map<String, NewPartitions> newPartitionsMap = new HashMap<>();
    newPartitionsMap.put(topic, newPartitions);
    CreatePartitionsResult result =
client.createPartitions(newPartitionsMap);
    result.all().get();

    client.close();
}
}
```

本章主要讲解了Kafka两大核心之一：主题，通过对主题的增删该查、配置等内容来了解主题的相关内容。

第5章 分区

tips 学完这一章你可以***

深入学习Kafka分区的管理

包括：优先副本的选举、分区重新分配等

Kafka可以将主题划分为多个分区（Partition），会根据分区规则选择把消息存储到哪个分区中，只要如果分区规则设置的合理，那么所有的消息将会被均匀的分布到不同的分区中，这样就实现了负载均衡和水平扩展。另外，多个订阅者可以从一个或者多个分区中同时消费数据，以支撑海量数据处理能力。

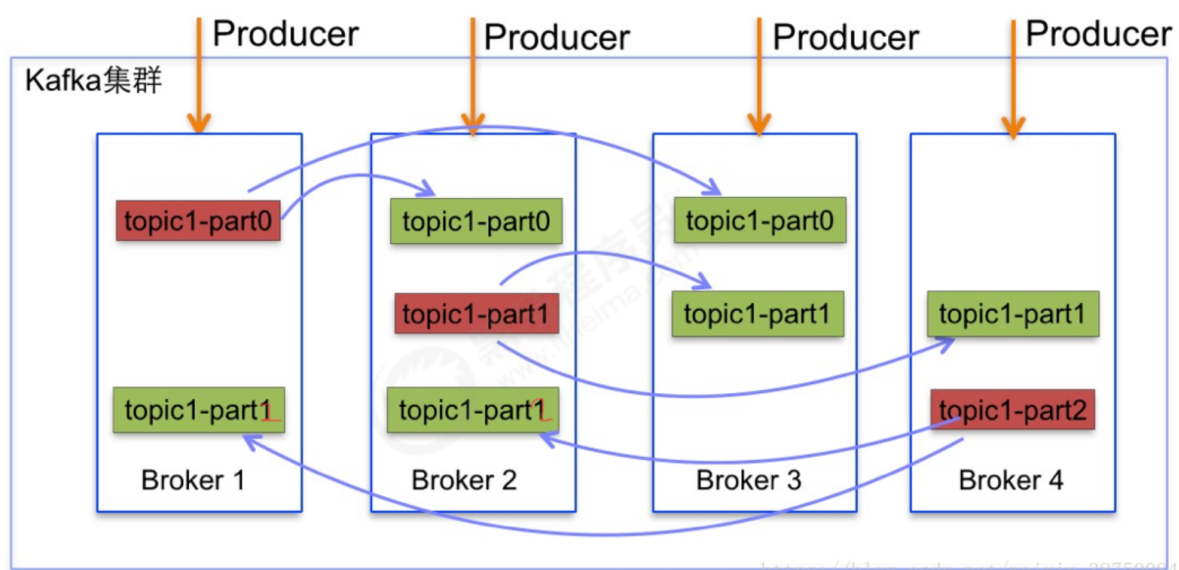
顺便说一句，由于消息是以追加到分区中的，多个分区顺序写磁盘的总效率要比随机写内存还要高（引用Apache Kafka - A High Throughput Distributed Messaging System的观点），是Kafka高吞吐率的重要保证之一。

5.1 副本机制

由于Producer和Consumer都只会与Leader角色的分区副本相连，所以kafka需要以集群的组织形式提供主题下的消息高可用。kafka支持主备复制，所以消息具备高可用和持久性。

一个分区可以有多个副本，这些副本保存在不同的broker上。每个分区的副本中都会有一个作为Leader。当一个broker失败时，Leader在这台broker上的分区都会变得不可用，kafka会自动移除Leader，再其他副本中选一个作为新的Leader。

在通常情况下，增加分区可以提供kafka集群的吞吐量。然而，也应该意识到集群的总分区数或是单台服务器上的分区数过多，会增加不可用及延迟的风险。



```
zookeeper localhost:2181 Topic: heima
// heima这个主题有三个分区，一个副本同处在一个节点当中
Topic:heima      PartitionCount:3      ReplicationFactor:1      Configs:
Topic: heima     Partition: 0      Leader: 0      Replicas: 0      Isr: 0
Topic: heima     Partition: 1      Leader: 0      Replicas: 0      Isr: 0
Topic: heima     Partition: 2      Leader: 0      Replicas: 0      Isr: 0
```

5.2 分区Leader选举

可以预见的是,如果某个分区的Leader挂了,那么其它跟随者将会进行选举产生一个新的leader,之后所有的读写就会转移到这个新的Leader上,在kafka中,其不是采用常见的多数选举的方式进行副本的Leader选举,而是会在Zookeeper上针对每个Topic维护一个称为ISR (in-sync replica , 已同步的副本) 的集合,显然还有一些副本没有来得及同步。只有这个ISR列表里面的才有资格成为leader(先使用ISR里面的第一个, 如果不行依次类推, 因为ISR里面的是同步副本, 消息是最完整且各个节点都是一样的)。通过ISR,kafka需要的冗余度较低, 可以容忍的失败数比较高。假设某个topic有f+1个副本, kafka可以容忍f个不可用,当然,如果全部ISR里面的副本都不可用,也可以选择其他可用的副本,只是存在数据的不一致。

5.3 分区重新分配

我们往已经部署好的Kafka集群里面添加机器是最正常不过的需求, 而且添加起来非常地方便, 我们需要做的是从已经部署好的Kafka节点中复制相应的配置文件, 然后把里面的broker id修改成全局唯一的, 最后启动这个节点即可将它加入到现有Kafka集群中。

但是问题来了, 新添加的Kafka节点并不会自动地分配数据, 所以无法分担集群的负载, 除非我们新建一个topic。但是现在我们想手动将部分分区移到新添加的Kafka节点上, Kafka内部提供了相关的工具来重新分布某个topic的分区。

具体步骤

- 第一步：我们创建一个有三个节点的集群，详情可查看第九章集群的搭建

```
itcast@Server-node:/mnt/d/kafka-cluster/kafka-1$ bin/kafka-topics.sh --create --
zookeeper localhost:2181 --topic heima-par --partitions 3 --replication-factor 3
Created topic heima-par.
```

详情查看

```
itcast@Server-node:/mnt/d/kafka-cluster/kafka-1$ bin/kafka-topics.sh --describe
--zookeeper localhost:2181 --topic heima
-par
Topic:heima-par PartitionCount:3      ReplicationFactor:3      Configs:
Topic: heima-par Partition: 0      Leader: 2      Replicas: 2,1,0 Isr:
2,1,0
Topic: heima-par Partition: 1      Leader: 0      Replicas: 0,2,1 Isr: 0
Topic: heima-par Partition: 2      Leader: 1      Replicas: 1,0,2 Isr:
1,0,2
itcast@Server-node:/mnt/d/kafka-cluster/kafka-1$
```

从上面的输出可以看出heima-par这个主题一共有三个分区, 有三个副本

- 第二步：主题heima-par再添加一个分区



```
zookeeper localhost:2181 --topic heima-par
```

```
r --partitions 4
```

```
WARNING: If partitions are increased for a topic that has a key, the partition
logic or ordering of the messages will be affected
Adding partitions succeeded!
```

查看详情已经变成4个分区

```
itcast@Server-node:/mnt/d/kafka-cluster/kafka-1$ bin/kafka-topics.sh --describe
--zookeeper localhost:2181 --topic heima
-par
Topic:heima-par PartitionCount:4      ReplicationFactor:3      Configs:
Topic: heima-par      Partition: 0      Leader: 2      Replicas: 2,1,0 Isr:
2,1,0
Topic: heima-par      Partition: 1      Leader: 0      Replicas: 0,2,1 Isr: 0
Topic: heima-par      Partition: 2      Leader: 1      Replicas: 1,0,2 Isr:
1,0,2
Topic: heima-par      Partition: 3      Leader: 2      Replicas: 2,1,0 Isr:
2,1,0
```

这样会导致broker2维护更多的分区

- 第三步：再添加一个broker节点

查看主题信息

```
itcast@Server-node:/mnt/d/kafka-cluster/kafka-1$ bin/kafka-topics.sh --describe
-zookeeper localhost:2181 --topic heima-par
Topic:heima-par PartitionCount:4      ReplicationFactor:3      Configs:
Topic: heima-par      Partition: 0      Leader: 2      Replicas: 2,1,0 Isr:
2,1,0
Topic: heima-par      Partition: 1      Leader: 0      Replicas: 0,2,1 Isr: 0
Topic: heima-par      Partition: 2      Leader: 1      Replicas: 1,0,2 Isr:
1,0,2
Topic: heima-par      Partition: 3      Leader: 2      Replicas: 2,1,0 Isr:
2,1,0
```

从上面输出信息可以看出新添加的节点并没有分配之前主题的分区

- 第四步：重新分配

现在我们需要将原先分布在broker 1-3节点上的分区重新分布到broker 1-4节点上，借助kafka-reassign-partitions.sh工具生成reassign plan，不过我们先得按照要求定义一个文件，里面说明哪些topic需要重新分区，文件内容如下：

```
itcast@Server-node:/mnt/d/kafka-cluster/kafka-1$ cat reassign.json
{"topics":[{"topic":"heima-par"}],
 "version":1
}
```

然后使用 kafka-reassign-partitions.sh 工具生成reassign plan

```
bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --topics-to -move-json-file
reassign.json --broker-list "0,1,2,3" --generate
```



```
partition.sh zookeeper localhost:2181 --move-json-file reassign.json --broker-list "0,1,2,3" --generate
Current partition replica assignment
{"version":1,"partitions":[{"topic":"heima-par","partition":2,"replicas":
[1,0,2],"log_dirs":["any","any","any"]},{ "topic":"heima-
par","partition":1,"replicas":[0,2,1],"log_dirs":["any","any","any"]},
{"topic":"heima-par","partition":0,"replicas":[2,1,0],"log_dirs":
["any","any","any"]},{ "topic":"heima-par","partition":3,"replicas":
[2,1,0],"log_dirs":["any","any","any"]}]}

Proposed partition reassignment configuration
{"version":1,"partitions":[{"topic":"heima-par","partition":0,"replicas":
[1,2,3],"log_dirs":["any","any","any"]},{ "topic":"heima-
par","partition":2,"replicas":[3,0,1],"log_dirs":["any","any","any"]},
{"topic":"heima-par","partition":1,"replicas":[2,3,0],"log_dirs":
["any","any","any"]},{ "topic":"heima-par","partition":3,"replicas":
[0,1,2],"log_dirs":["any","any","any"]}]}

```

上面命令中

--generate 表示指定类型参数

--topics-to-move-json-file 指定分区重分配对应的主题清单路径

注意：

命令输入两个json字符串，第一个JSON内容为当前的分区副本分配情况，第二个为重新分配的候选方案，注意这里只是生成一份可行性的方案，并没有真正执行重分配的动作。

我们将第二个JSON内容保存到名为result.json文件里面（文件名不重要，文件格式也不一定要以json为结尾，只要保证内容是json即可），然后执行这些reassign plan：

重新分配JSON文件

```
{
  "version": 1,
  "partitions": [
    {
      "topic": "heima-par",
      "partition": 0,
      "replicas": [
        1,
        2,
        3
      ],
      "log_dirs": [
        "any",
        "any",
        "any"
      ]
    },
    {
      "topic": "heima-par",
      "partition": 2,
      "replicas": [
        3,
        0,

```




```
        "log_dirs": [
            "any",
            "any",
            "any"
        ]
    },
    {
        "topic": "heima-par",
        "partition": 1,
        "replicas": [
            2,
            3,
            0
        ],
        "log_dirs": [
            "any",
            "any",
            "any"
        ]
    },
    {
        "topic": "heima-par",
        "partition": 3,
        "replicas": [
            0,
            1,
            2
        ],
        "log_dirs": [
            "any",
            "any",
            "any"
        ]
    }
]
}
```

执行分配策略

```
itcast@Server-node:/mnt/d/kafka-cluster/kafka-1$ bin/kafka-reassign-
partitions.sh --zookeeper localhost:2181 --reassignment-
ent-json-file result.json --execute
Current partition replica assignment
```

```
{"version":1,"partitions":[{"topic":"heima-par","partition":2,"replicas":
[1,0,2],"log_dirs":["any","any","any"]},{topic":"heima-
par","partition":1,"replicas":[0,2,1],"log_dirs":["any","any","any"]},
{"topic":"heima-par","partition":0,"replicas":[2,1,0],"log_dirs":
["any","any","any"]},{topic":"heima-par","partition":3,"replicas":
[2,1,0],"log_dirs":["any","any","any"]}]}
```

Save this to use as the --reassignment-json-file option during rollback
Successfully started reassignment of partitions.


```
partition.sh --zookeeper localhost:2181 --reassignment-json-file replication-factor.json --verify
Status of partition reassignment:
Reassignment of partition heima-par-3 completed successfully
Reassignment of partition heima-par-0 is still in progress
Reassignment of partition heima-par-2 is still in progress
Reassignment of partition heima-par-1 is still in progress
```

从上面信息可以看出 heima-par-3已经完成，其他三个正在进行中。

5.4 修改副本因子

场景

实际项目中我们可能在创建topic时没有设置好正确的replication-factor，导致kafka集群虽然是高可用的，但是该topic在有broker宕机时，可能发生无法使用的情况。topic一旦使用又不能轻易删除重建，因此动态增加副本因子就成为最终的选择。

说明：kafka 1.0版本配置文件默认没有default.replication.factor=x，因此如果创建topic时，不指定-replication-factor 想，默认副本因子为1。我们可以在自己的server.properties中配置上常用的副本因子，省去手动调整。例如设置default.replication.factor=3

首先我们配置topic的副本，保存为json文件

```
{
  "version":1,
  "partitions":[
    {"topic":"heima","partition":0,"replicas":[0,1,2]},
    {"topic":"heima","partition":1,"replicas":[0,1,2]},
    {"topic":"heima","partition":2,"replicas":[0,1,2]}
  ]
}
```

然后执行脚本 bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --reassignment-json-file replication-factor.json --execute

```
itcast@Server-node:/mnt/d/kafka_2.12-2.2.1$ bin/kafka-reassign-partitions.sh --
zookeeper localhost:2181 --reassignment-json-file replication-factor.json --
execute
Current partition replica assignment

{"version":1,"partitions":[{"topic":"topic0703","partition":1,"replicas":
[1,0],"log_dirs":["any","any"]},{"topic":"topic0703","partition":0,"replicas":
[0,1],"log_dirs":["any","any"]},{"topic":"topic0703","partition":2,"replicas":
[2,0],"log_dirs":["any","any"]}]}

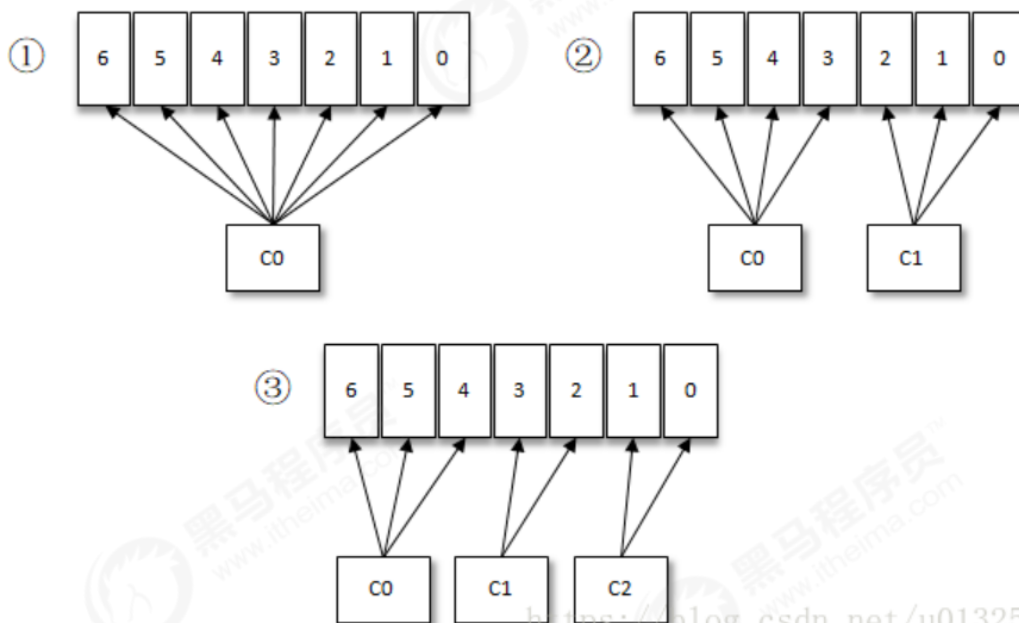
Save this to use as the --reassignment-json-file option during rollback
Successfully started reassignment of partitions.
```

验证

```
zookeeper:localhost:2181 topic:topic0703
Topic:topic0703 PartitionCount:3 ReplicationFactor:3 Configs:
Topic: topic0703 Partition: 0 Leader: 0 Replicas: 0,1,2 Isr: 0
Topic: topic0703 Partition: 1 Leader: 1 Replicas: 0,1,2 Isr: 1,0
Topic: topic0703 Partition: 2 Leader: 2 Replicas: 0,1,2 Isr: 2,0
```

5.5 分区分配策略

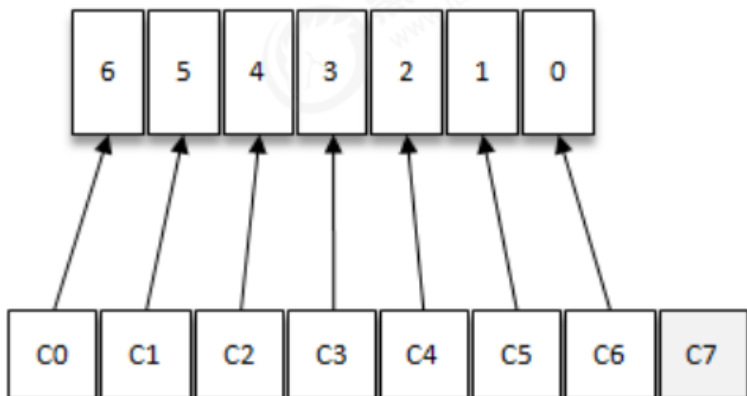
按照Kafka默认的消费逻辑设定，一个分区只能被同一个消费组（ConsumerGroup）内的一个消费者消费。假设目前某消费组内只有一个消费者C0，订阅了一个topic，这个topic包含7个分区，也就是说这个消费者C0订阅了7个分区，参考下图



此时消费组内又加入了一个新的消费者C1，按照既定的逻辑需要将原来消费者C0的部分分区分配给消费者C1消费，情形上图（2），消费者C0和C1各自负责消费所分配到的分区，相互之间并无实质性的干扰。

接着消费组内又加入了一个新的消费者C2，如此消费者C0、C1和C2按照上图（3）中的方式各自负责消费所分配到的分区。

如果消费者过多，出现了消费者的数量大于分区的数量的情况，就会有消费者分配不到任何分区。参考下图，一共有8个消费者，7个分区，那么最后的消费者C7由于分配不到任何分区进而就无法消费任何消息。





数的值为：`org.apache.kafka.clients.consumer.RangeAssignor`，即采用`RangeAssignor`分配策略。除此之外，Kafka中还提供了另外两种分配策略：`RoundRobinAssignor`和`StickyAssignor`。消费者客户端参数`partition.assignment.strategy`可以配置多个分配策略，彼此之间以逗号分隔。

RangeAssignor分配策略

参考源码：`org.apache.kafka.clients.consumer.RangeAssignor`

`RangeAssignor`策略的原理是按照消费者总数和分区总数进行整除运算来获得一个跨度，然后将分区按照跨度进行平均分配，以保证分区尽可能均匀地分配给所有的消费者。对于每一个topic，`RangeAssignor`策略会将消费组内所有订阅这个topic的消费者按照名称的字典序排序，然后为每个消费者划分固定的分区范围，如果不够平均分配，那么字典序靠前的消费者会被多分配一个分区。

假设 $n = \text{分区数} / \text{消费者数量}$ ， $m = \text{分区数} \% \text{消费者数量}$ ，那么前 m 个消费者每个分配 $n+1$ 个分区，后面的 $(\text{消费者数量} - m)$ 个消费者每个分配 n 个分区。

假设消费组内有2个消费者C0和C1，都订阅了主题t0和t1，并且每个主题都有4个分区，那么所订阅的所有分区可以标识为：`t0p0`、`t0p1`、`t0p2`、`t0p3`、`t1p0`、`t1p1`、`t1p2`、`t1p3`。最终的分配结果为：

消费者C0: `t0p0`、`t0p1`、`t1p0`、`t1p1`
消费者C1: `t0p2`、`t0p3`、`t1p2`、`t1p3`

假设上面例子中2个主题都只有3个分区，那么所订阅的所有分区可以标识为：`t0p0`、`t0p1`、`t0p2`、`t1p0`、`t1p1`、`t1p2`。最终的分配结果为：

消费者C0: `t0p0`、`t0p1`、`t1p0`、`t1p1`
消费者C1: `t0p2`、`t1p2`

可以明显的看到这样的分配并不均匀，如果将类似的情形扩大，有可能会出现部分消费者过载的情况。

RoundRobinAssignor分配策略

参考源码：`org.apache.kafka.clients.consumer.RoundRobinAssignor`

`RoundRobinAssignor`策略的原理是将消费组内所有消费者以及消费者所订阅的所有topic的partition按照字典序排序，然后通过轮询方式逐个将分区以此分配给每个消费者。`RoundRobinAssignor`策略对应的`partition.assignment.strategy`参数值为：

`org.apache.kafka.clients.consumer.RoundRobinAssignor`。

假设消费组中有2个消费者C0和C1，都订阅了主题t0和t1，并且每个主题都有3个分区，那么所订阅的所有分区可以标识为：`t0p0`、`t0p1`、`t0p2`、`t1p0`、`t1p1`、`t1p2`。最终的分配结果为：

消费者C0: `t0p0`、`t0p2`、`t1p1`
消费者C1: `t0p1`、`t1p0`、`t1p2`

如果同一个消费组内的消费者所订阅的信息是不相同的，那么在执行分区分配的时候就不是完全的轮询分配，有可能会导致分区分配的不均匀。如果某个消费者没有订阅消费组内的某个topic，那么在分配分区的时候此消费者将分配不到这个topic的任何分区。

假设消费组内有3个消费者C0、C1和C2，它们共订阅了3个主题：`t0`、`t1`、`t2`，这3个主题分别有1、2、3个分区，即整个消费组订阅了`t0p0`、`t1p0`、`t1p1`、`t2p0`、`t2p1`、`t2p2`这6个分区。具体而言，消费者C0订阅的是主题t0，消费者C1订阅的是主题t0和t1，消费者C2订阅的是主题t0、t1和t2，那么最终的分配结果为：



消费者C2: t1p1、t2p0、t2p1、t2p2

可以看到RoundRobinAssignor策略也不是十分完美，这样分配其实并不是最优解，因为完全可以将分区t1p1分配给消费者C1。

StickyAssignor分配策略

参考源码：org.apache.kafka.clients.consumer.StickyAssignor

Kafka从0.11.x版本开始引入这种分配策略，它主要有两个目的：

分区的分配要尽可能的均匀；分区的分配尽可能的与上次分配的保持相同。

当两者发生冲突时，第一个目标优先于第二个目标。鉴于这两个目标，StickyAssignor策略的具体实现要比RangeAssignor和RoundRobinAssignor这两种分配策略要复杂很多。

假设消费组内有3个消费者：C0、C1和C2，它们都订阅了4个主题：t0、t1、t2、t3，并且每个主题有2个分区，也就是说整个消费组订阅了t0p0、t0p1、t1p0、t1p1、t2p0、t2p1、t3p0、t3p1这8个分区。最终的分配结果如下：

消费者C0: t0p0、t1p1、t3p0
消费者C1: t0p1、t2p0、t3p1
消费者C2: t1p0、t2p1

假设此时消费者C1脱离了消费组，那么消费组就会执行再平衡操作，进而消费分区会重新分配。如果采用RoundRobinAssignor策略，那么此时的分配结果如下：

消费者C0: t0p0、t1p0、t2p0、t3p0
消费者C2: t0p1、t1p1、t2p1、t3p1

如分配结果所示，RoundRobinAssignor策略会按照消费者C0和C2进行重新轮询分配。而如果此时使用的是StickyAssignor策略，那么分配结果为：

消费者C0: t0p0、t1p1、t3p0、t2p0
消费者C2: t1p0、t2p1、t0p1、t3p1

可以看到分配结果中保留了上一次分配中对于消费者C0和C2的所有分配结果，并将原来消费者C1的“负担”分配给了剩余的两个消费者C0和C2，最终C0和C2的分配还保持了均衡。

自定义分配策略

需实现：org.apache.kafka.clients.consumer.internals.PartitionAssignor

继承自：org.apache.kafka.clients.consumer.internals.AbstractPartitionAssignor

总结

本章讲解了对分区及副本的一系列操作，如分区副本机制、分区重新分配、修改副本因子等。

第6张 Kafka存储

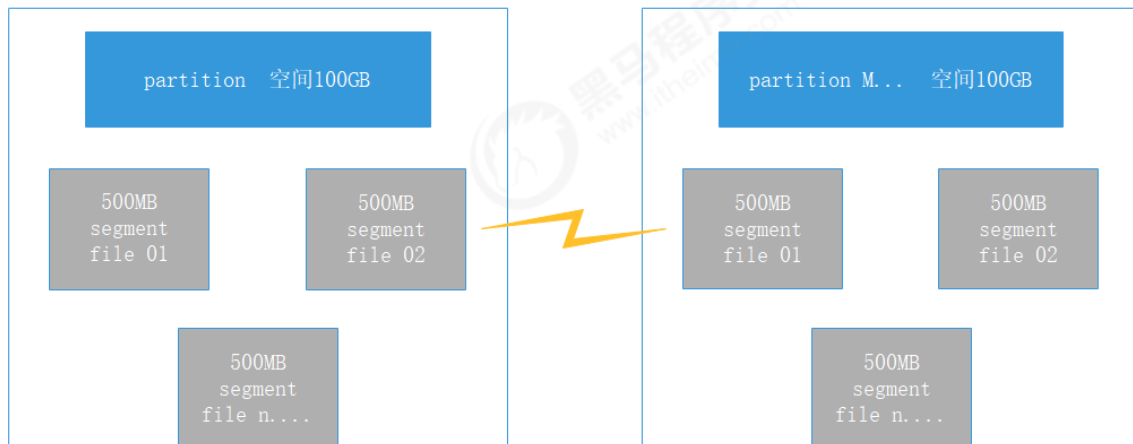
在完成Kafka应用开发的基础上，知道文件存储机制

Kafka为什么使用磁盘作为存储介质

分析文件存储格式

快速检索消息

6.1 存储结构概述



- 每一个partition(文件夹)相当于一个巨型文件被平均分配到多个大小相等segment(段)数据文件里。但每一个段segment file消息数量不一定相等，这样的特性方便old segment file高速被删除。（默认情况下每一个文件大小为1G）
- 每一个partition仅仅须要支持顺序读写即可了。segment文件生命周期由服务端配置参数决定。

partition中segment文件存储结构

segment file组成：由2大部分组成。分别为index file和数据file，此2个文件——相应，成对出现，后缀“.index”和“.log”分别表示为segment索引文件、数据文件。

segment文件命名规则：partition全局的第一个segment从0开始，兴许每一个segment文件名称为上一个segment文件最后一条消息的offset值。

数值最大为64位long大小。19位数字字符长度，没有数字用0填充。

```
itcast@Server-node:/mnt/d/kafka_2.12-2.2.1$ ll /tmp/kafka/log/heimu-0/
total 20480
drwxr-xr-x 1 itcast sudo          512 Aug 29 09:38 ./
drwxrwxrwx 1 dayuan dayuan       512 Aug 29 09:41 ../
-rw-r--r-- 1 itcast sudo    10485760 Aug 29 09:38 00000000000000000000.index
-rw-r--r-- 1 itcast sudo           0 Aug 29 09:38 00000000000000000000.log
-rw-r--r-- 1 itcast sudo    10485756 Aug 29 09:38 00000000000000000000.timeindex
-rw-r--r-- 1 itcast sudo           8 Aug 29 09:38 leader-epoch-checkpoint
itcast@Server-node:/mnt/d/kafka_2.12-2.2.1$
```

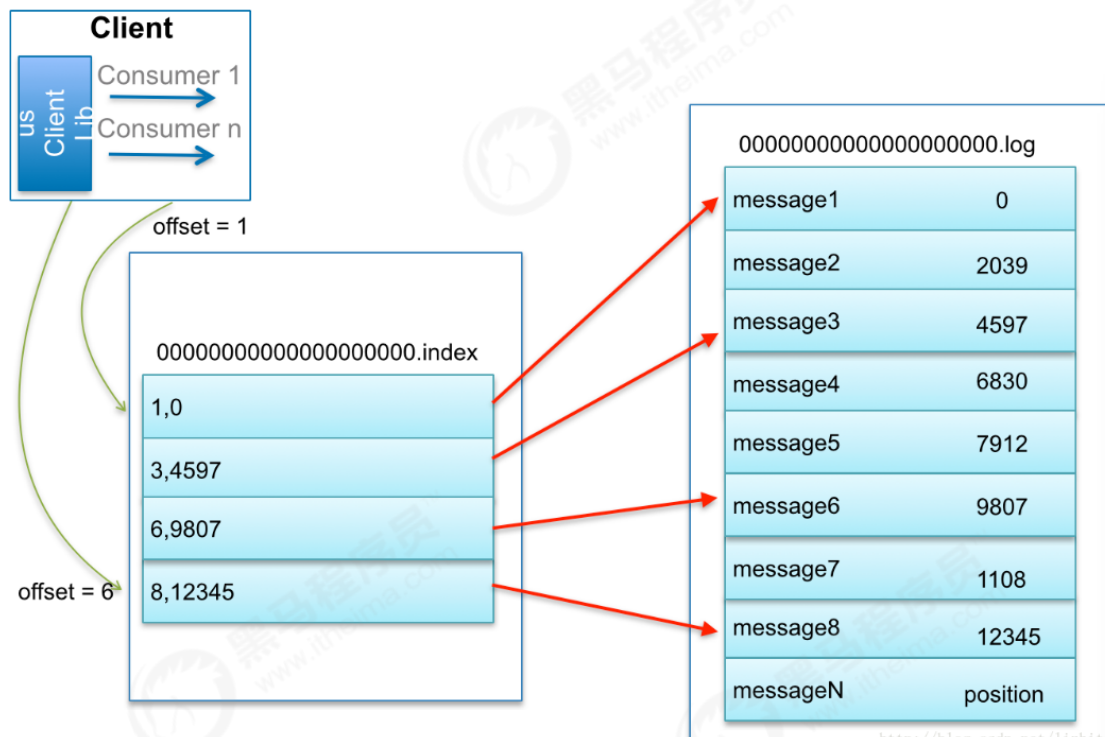
6.2 日志索引

6.2.1 数据文件的分段

面，数据文件以该段中最小的offset命名。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。

6.2.2 偏移量索引

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。



比如：要查找绝对offset为7的Message：

首先是用二分查找确定它是在哪个LogSegment中，自然是在第一个Segment中。打开这个Segment的index文件，也是用二分查找找到offset小于或者等于指定offset的索引条目中最大的那个offset。自然offset为6的那个索引是我们要找的，通过索引文件我们知道offset为6的Message在数据文件中的位置为9807。

打开数据文件，从位置为9807的那个地方开始顺序扫描直到找到offset为7的那条Message。

这套机制是建立在offset是有序的。索引文件被映射到内存中，所以查找的速度还是很快的。

一句话，Kafka的Message存储采用了分区(partition)，分段(LogSegment)和稀疏索引这几个手段来达到了高效性。

6.3 日志清理

6.3.1 日志删除

Kafka日志管理器允许定制删除策略。目前的策略是删除修改时间在N天之前的日志（按时间删除），也可以使用另外一个策略：保留最后的N GB数据的策略(按大小删除)。为了避免在删除时阻塞读操作，采用了copy-on-write形式的实现，删除操作进行时，读取操作的二分查找功能实际是在一个静态的快照副本上进行的，这类似于Java的CopyOnWriteArrayList。Kafka消费日志删除思想：Kafka把topic中一个partition大文件分成多个小文件段，通过多个小文件段，就容易定期清除或删除已经消费完文件，减少磁盘占用

清理超过指定时间清理:

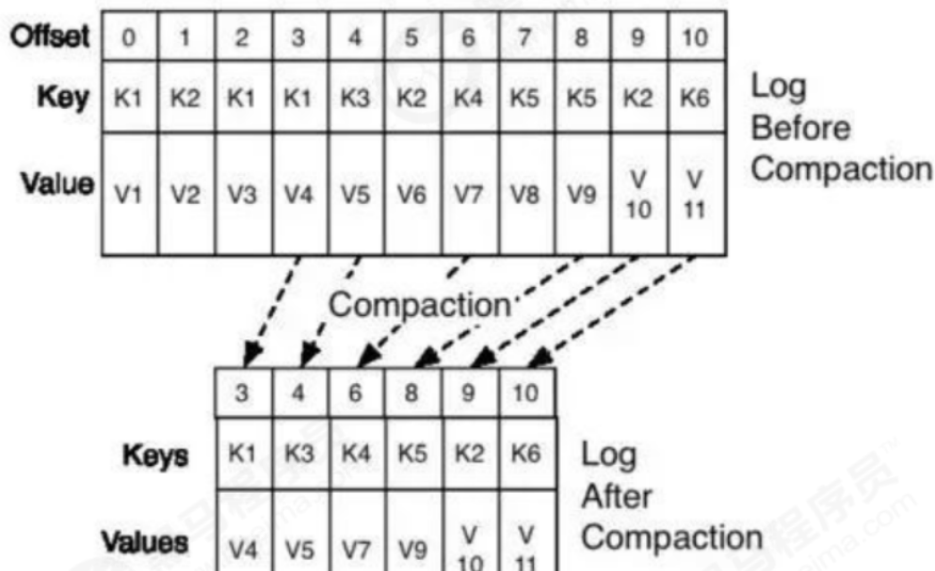
`log.retention.hours=16`

超过指定大小后，删除旧的消息:

`log.retention.bytes=1073741824`

6.3.2 日志压缩

将数据压缩，只保留每个key最后一个版本的数据。首先在broker的配置中设置 `log.cleaner.enable=true` 启用cleaner，这个默认是关闭的。在Topic的配置中设置 `log.cleanup.policy=compact` 启用压缩策略。



压缩后的offset可能是不连续的，比如上图中没有5和7，因为这些offset的消息被merge了，当从这些offset消费消息时，将会拿到比这个offset大的offset对应的消息，比如，当试图获取offset为5的消息时，实际上会拿到offset为6的消息，并从这个位置开始消费。

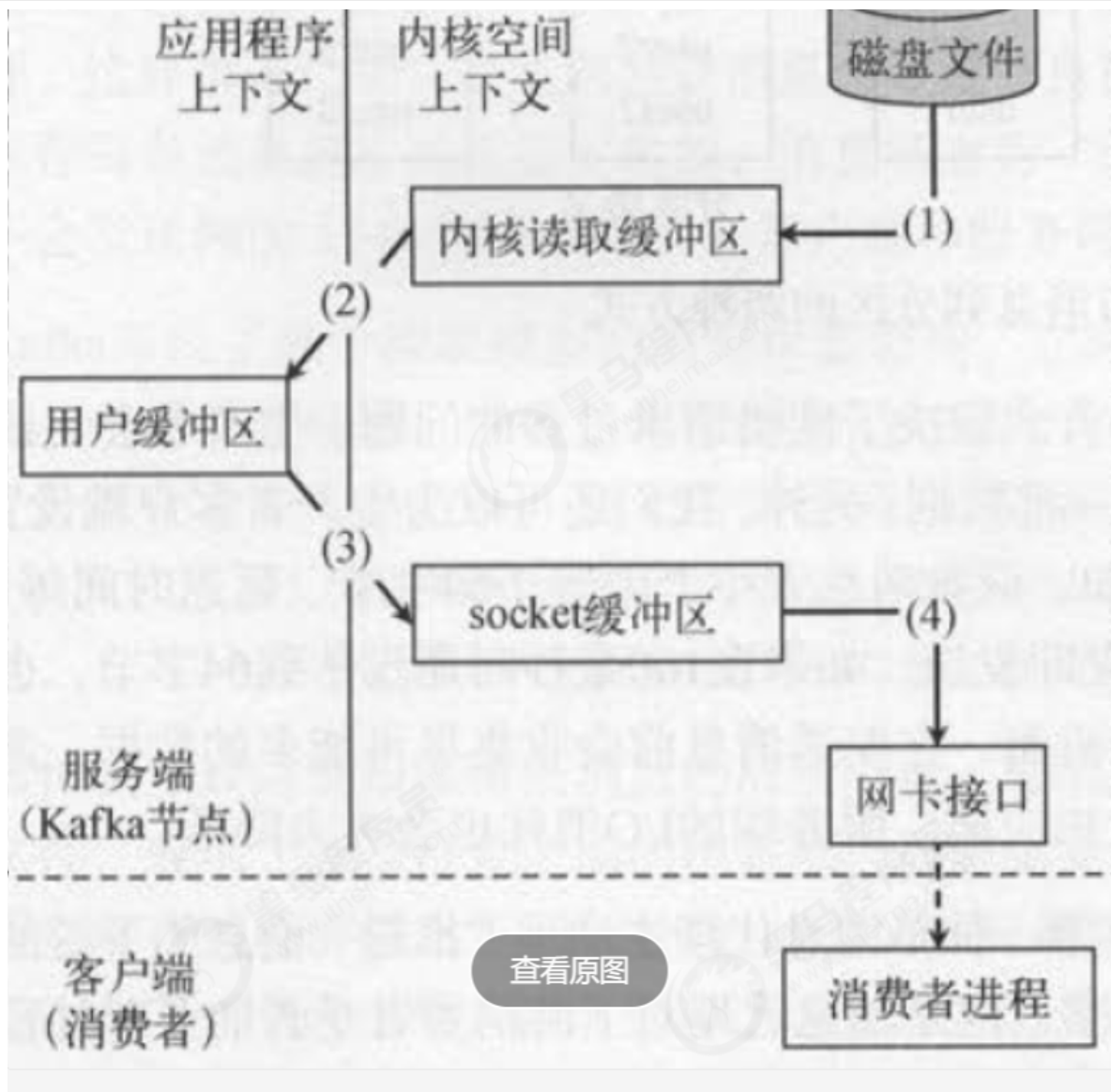
这种策略只适合特俗场景，比如消息的key是用户ID，消息体是用户的资料，通过这种压缩策略，整个消息集里就保存了所有用户最新的资料。

压缩策略支持删除，当某个Key的最新版本的消息没有内容时，这个Key将被删除，这也符合以上逻辑。

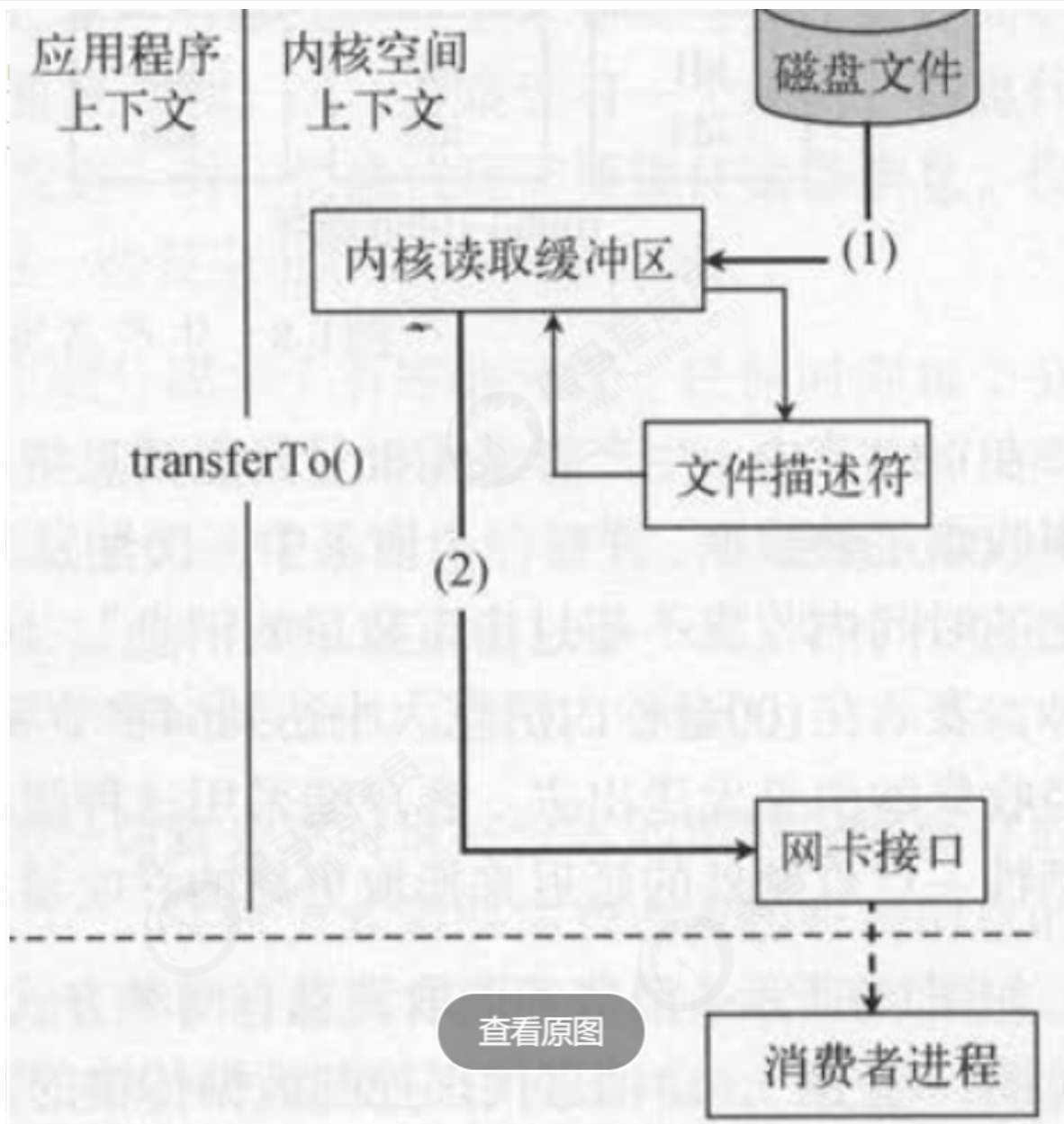
6.4 磁盘存储优势

Kafka在设计的时候，采用了文件追加的方式来写入消息，即只能在日志文件的尾部追加新的消息，并且不允许修改已经写入的消息，这种方式属于典型的顺序写入此判断的操作，所以就算是Kafka使用磁盘作为存储介质，所能实现的吞吐量也非常可观。

Kafka中大量使用页缓存，这页是Kafka实现高吞吐的重要因素之一。



除了消息顺序追加，页缓存等技术，Kafka还使用了零拷贝技术来进一步提升性能。“零拷贝技术”只用将磁盘文件的数据复制到页面缓存中一次，然后将数据从页面缓存直接发送到网络中（发送给不同的订阅者时，都可以使用同一个页面缓存），避免了重复复制操作。如果有10个消费者，传统方式下，数据复制次数为 $4 \times 10 = 40$ 次，而使用“零拷贝技术”只需要 $1 + 10 = 11$ 次，一次为从磁盘复制到页面缓存，10次表示10个消费者各自读取一次页面缓存。



总结

本章主要讲述了Kafka中与存储相关的知识点，包含了Kafka自身的日志格式、日志索引、日志清理等方面的内容，也涉及到底层物理存储相关的知识。通过本章的学习，可以Kafka核心机理有较深入的认知。

第7章 稳定性

*tips 学完这一章你可以

深入学习Kafka在保证高性能、高吞吐的同时通过各种机制来保证高可用性

络问题而造成通信中断，那producer就无法判断该条消息是否已经提交（commit）。虽然Kafka无法确定网络故障期间发生了什么，但是producer可以retry多次，确保消息已经正确传输到broker中，所以目前Kafka实现的是at least once。

7.1 幂等性

场景

所谓幂等性，就是对接口的多次调用所产生的结果和调用一次是一致的。生产者在进行重试的时候有可能会重复写入消息，二使用Kafka的幂等性功能就可以避免这种情况。

幂等性是有条件的：

- 只能保证 Producer 在单个会话内不丢不重，如果 Producer 出现意外挂掉再重启是无法保证的（幂等性情况下，是无法获取之前的状态信息，因此是无法做到跨会话级别的不丢不重）；
- 幂等性不能跨多个 Topic-Partition，只能保证单个 partition 内的幂等性，当涉及多个 Topic-Partition 时，这中间的状态并没有同步。

Producer 使用幂等性的示例非常简单，与正常情况下 Producer 使用相比变化不大，只需要把 Producer 的配置 enable.idempotence 设置为 true 即可，如下所示：

```
Properties props = new Properties();
props.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, "true");
props.put("acks", "all"); // 当 enable.idempotence 为 true, 这里默认为 all
props.put("bootstrap.servers", "localhost:9092");
props.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");

KafkaProducer producer = new KafkaProducer(props);

producer.send(new ProducerRecord(topic, "test");
```

7.2 事务

场景

幂等性并不能跨多个分区运作，而事务可以弥补这个缺憾，事务可以保证对多个分区写入操作的原子性。操作的原子性是指多个操作要么全部成功，要么全部失败，不存在部分成功部分失败的可能。

为了实现事务，应用程序必须提供唯一的transactionId，这个参数通过客户端程序来进行设定。

见代码库：com.heima.kafka.chapter7.ProducerTransactionSend

```
properties.put(ProducerConfig.TRANSACTIONAL_ID_CONFIG, transactionId);
```

前期准备

事务要求生产者开启幂等性特性，因此通过将transactional.id参数设置为非空从而开启事务特性的同时需要将ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG设置为true（默认值为true），如果显示设置为false，则会抛出异常。

KafkaProducer提供了5个与事务相关的方法，具体如下：



```
public void initTransactions()
//开启事务
public void beginTransaction()
//为消费者提供事务内的位移提交操作
public void sendOffsetsToTransaction(Map<TopicPartition, OffsetAndMetadata>
offsets, String consumerGroupId)
//提交事务
public void commitTransaction()
//终止事务，类似于回滚
public void abortTransaction()
```

案例解析

见代码库：com.heimakafka.chapter7.ProducerTransactionSend

消息发送端

```
/**
 * Kafka Producer事务的使用
 */
public class ProducerTransactionSend {
    public static final String topic = "topic-transaction";
    public static final String brokerList = "localhost:9092";
    public static final String transactionId = "transactionId";

    public static void main(String[] args) {
        Properties properties = new Properties();
        properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
            StringSerializer.class.getName());
        properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
            StringSerializer.class.getName());
        properties.put(ProducerConfig.BootstrapServersConfig, brokerList);
        properties.put(ProducerConfig.TransactionIdConfig, transactionId);

        KafkaProducer<String, String> producer = new KafkaProducer<>
            (properties);

        producer.initTransactions();
        producer.beginTransaction();

        try {
            //处理业务逻辑并创建ProducerRecord
            ProducerRecord<String, String> record1 = new ProducerRecord<>(topic,
                "msg1");
            producer.send(record1);
            ProducerRecord<String, String> record2 = new ProducerRecord<>(topic,
                "msg2");
            producer.send(record2);
            ProducerRecord<String, String> record3 = new ProducerRecord<>(topic,
                "msg3");
            producer.send(record3);
            //处理一些其它逻辑
            producer.commitTransaction();
        } catch (ProducerFencedException e) {
            producer.abortTransaction();
        }
    }
}
```

```
}
```

模拟事务回滚案例

```
try {  
    //处理业务逻辑并创建ProducerRecord  
    ProducerRecord<String, String> record1 = new ProducerRecord<>(topic,  
    "msg1");  
    producer.send(record1);  
  
    //模拟事务回滚案例  
    System.out.println(1/0);  
  
    ProducerRecord<String, String> record2 = new ProducerRecord<>(topic,  
    "msg2");  
    producer.send(record2);  
    ProducerRecord<String, String> record3 = new ProducerRecord<>(topic,  
    "msg3");  
    producer.send(record3);  
    //处理一些其它逻辑  
    producer.commitTransaction();  
} catch (ProducerFencedException e) {  
    producer.abortTransaction();  
}
```

从上面案例中，msg1发送成功之后，出现了异常事务进行了回滚，则msg1消费端也收不到消息。

7.3 控制器

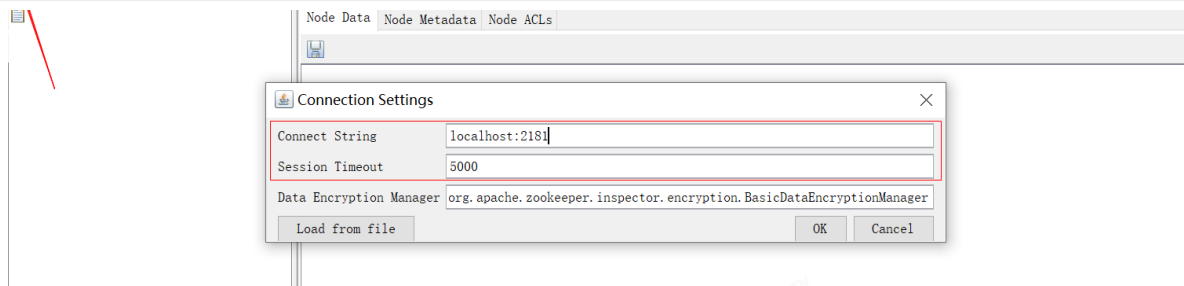
在Kafka集群中会有一个或者多个broker，其中有一个broker会被选举为控制器（Kafka Controller），它负责管理整个集群中所有分区和副本的状态。当某个分区的leader副本出现故障时，由控制器负责为该分区选举新的leader副本。当检测到某个分区的ISR集合发生变化时，由控制器负责通知所有broker更新其元数据信息。当使用kafka-topics.sh脚本为某个topic增加分区数量时，同样还是由控制器负责分区的重新分配。

Kafka中的控制器选举的工作依赖于Zookeeper，成功竞选为控制器的broker会在Zookeeper中创建/controller这个临时（EPHEMERAL）节点，此临时节点的内容参考如下：

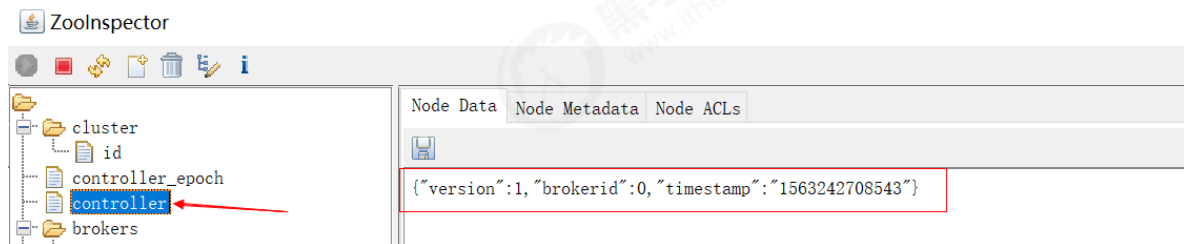
ZooInspector管理

- 使用zookeeper图形化的客户端工具(ZooInspector)提供的jar来进行管理，启动如下：

- 1、定位到jar所在目录
- 2、运行jar文件 `java -jar zookeeper-dev-ZooInspector.jar`
- 3、连接Zookeeper



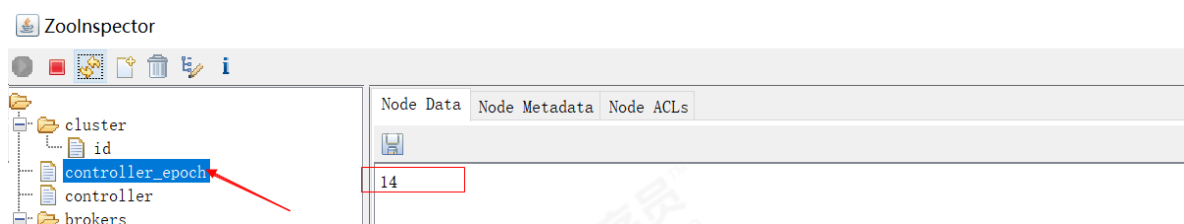
```
{"version":1,"brokerid":0,"timestamp":"1529210278988"}
```



其中version在目前版本中固定为1，brokerid表示称为控制器的broker的id编号，timestamp表示竞选称为控制器时的时间戳。

在任意时刻，集群中有且仅有一个控制器。每个broker启动的时候会去尝试去读取/controller节点的brokerid的值，如果读取到brokerid的值不为-1，则表示已经有其它broker节点成功竞选为控制器，所以当前broker就会放弃竞选；如果Zookeeper中不存在/controller这个节点，或者这个节点中的数据异常，那么就会尝试去创建/controller这个节点，当前broker去创建节点的时候，也有可能其他broker同时去尝试创建这个节点，只有创建成功的那个broker才会成为控制器，而创建失败的broker则表示竞选失败。每个broker都会在内存中保存当前控制器的brokerid值，这个值可以标识为activeControllerId。

Zookeeper中还有一个与控制器有关的/controller_epoch节点，这个节点是持久（PERSISTENT）节点，节点中存放的是一个整型的controller_epoch值。controller_epoch用于记录控制器发生变更的次数，即记录当前的控制器是第几代控制器，我们也可以称之为“控制器的纪元”。



controller_epoch的初始值为1，即集群中第一个控制器的纪元为1，当控制器发生变更时，没选出一个新的控制器就将该字段值加1。每个和控制器交互的请求都会携带上controller_epoch这个字段，如果请求的controller_epoch值小于内存中的controller_epoch值，则认为这个请求是向已经过期的控制器所发送的请求，那么这个请求会被认定为无效的请求。如果请求的controller_epoch值大于内存中的controller_epoch值，那么则说明已经有新的控制器当选了。由此可见，Kafka通过controller_epoch来保证控制器的唯一性，进而保证相关操作的一致性。

具备控制器身份的broker需要比其他普通的broker多一份职责，具体细节如下：

- 1、监听partition相关的变化。
- 2、监听topic相关的变化。
- 3、监听broker相关的变化

7.4 可靠性保证

1. 可靠性保证：确保系统在**各种不同的环境下**能够发生**一致**的行为
2. Kafka的保证
 - 保证**分区消息的顺序**
 - 如果使用**同一个生产者往同一个分区**写入消息，而且消息B在消息A之后写入
 - 那么Kafka可以保证消息B的偏移量比消息A的偏移量大，而且消费者会先读取消息A再读取消息B
 - 只有当消息被写入分区的所有同步副本时（文件系统缓存），它才被认为是已提交
 - 生产者可以选择接收不同类型的确认，控制参数 `acks`
 - 只要还有一个副本是活跃的，那么**已提交的消息就不会丢失**
 - **消费者只能读取已经提交的消息**

失效副本

怎么样判定一个分区是否有副本是处于同步失效状态的呢？从Kafka 0.9.x版本开始通过唯一的一个参数 `replica.lag.time.max.ms`（默认大小为10,000）来控制，当ISR中的一个follower副本滞后leader副本的时间超过参数`replica.lag.time.max.ms`指定的值时即判定为副本失效，需要将此follower副本剔除除ISR之外。具体实现原理很简单，当follower副本将leader副本的LEO（Log End Offset，每个分区最后一条消息的位置）之前的日志全部同步时，则认为该follower副本已经追赶上leader副本，此时更新该副本的`lastCaughtUpTimeMs`标识。Kafka的副本管理器（ReplicaManager）启动时会启动一个副本过期检测的定时任务，而这个定时任务会定时检查当前时间与副本的`lastCaughtUpTimeMs`差值是否大于参数`replica.lag.time.max.ms`指定的值。千万不要错误的认为follower副本只要拉取leader副本的数据就会更新`lastCaughtUpTimeMs`，试想当leader副本的消息流入速度大于follower副本的拉取速度时，follower副本一直不断的拉取leader副本的消息也不能与leader副本同步，如果还将此follower副本置于ISR中，那么当leader副本失效，而选取此follower副本为新的leader副本，那么就会有严重的消息丢失。

副本复制

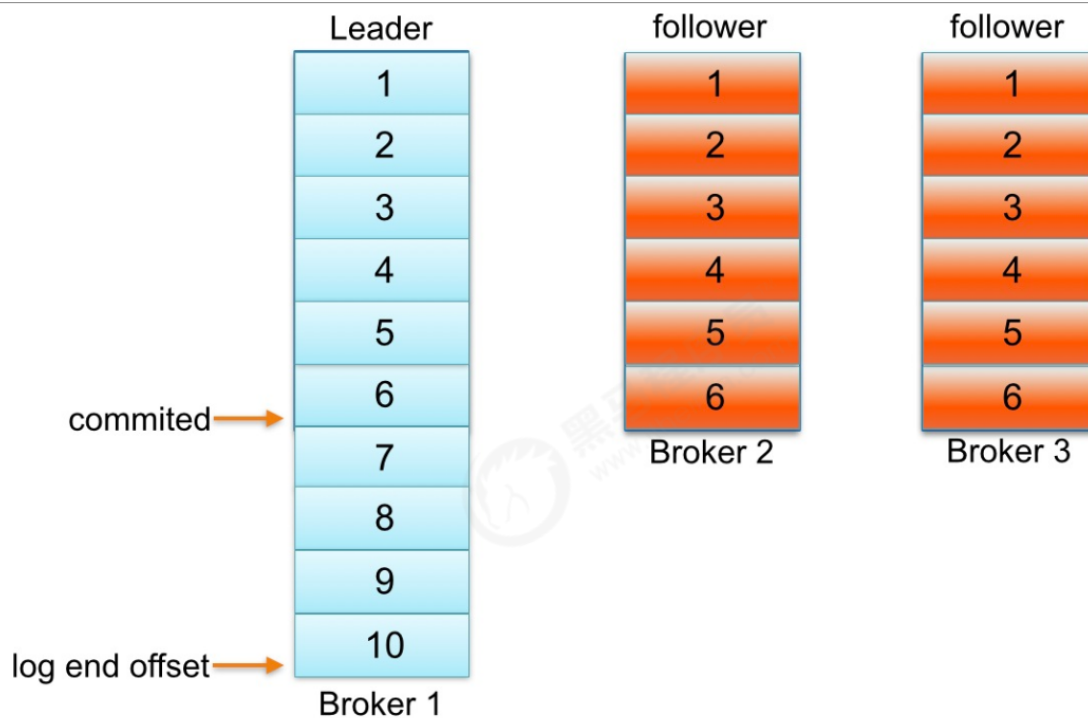
Kafka 中的每个主题分区都被复制了 n 次，其中的 n 是主题的复制因子（replication factor）。这允许Kafka 在集群服务器发生故障时自动切换到这些副本，以便在出现故障时消息仍然可用。Kafka 的复制是以分区为粒度的，分区的预写日志被复制到 n 个服务器。在 n 个副本中，一个副本作为 leader，其他副本成为 followers。顾名思义，producer 只能往 leader 分区上写数据（读也只能从 leader 分区上进行），followers 只按顺序从 leader 上复制日志。

一个副本可以不同步Leader有如下几个原因
慢副本：在一定周期时间内follower不能追赶上leader。最常见的原因之一是I/O瓶颈导致follower追加复制消息速度慢于从leader拉取速度。
卡住副本：在一定周期时间内follower停止从leader拉取请求。follower replica卡住了是由于GC暂停或follower失效或死亡。

新启动副本：当用户给主题增加副本因子时，新的follower不在同步副本列表中，直到他们完全赶上了leader日志。

如何确定副本是滞后的

`replica.lag.max.messages=4`



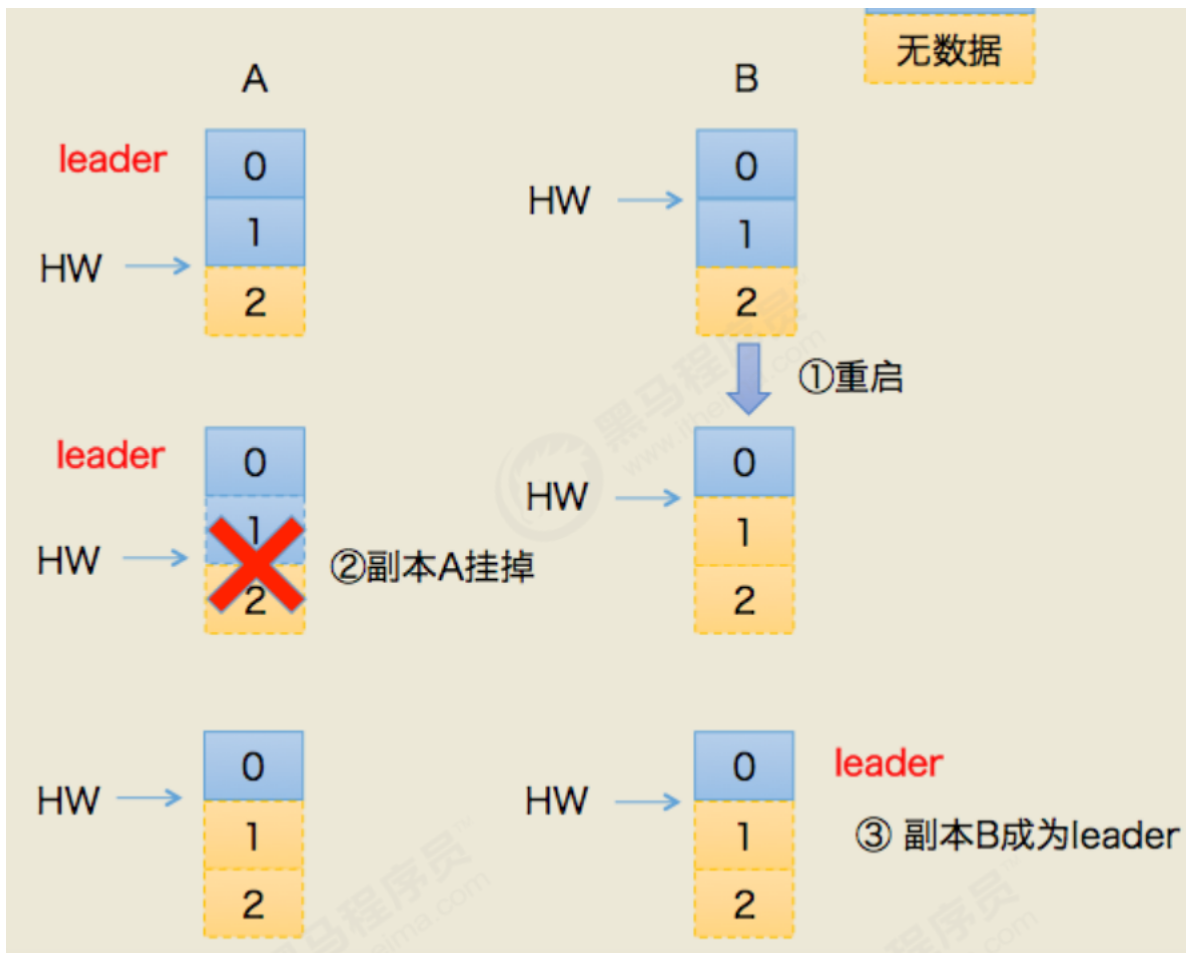
在服务端现在只有一个参数需要配置`replica.lag.time.max.ms`。这个参数解释replicas响应partition leader的最长等待时间。检测卡住或失败副本的探测——如果一个replica失败导致发送拉取请求时间间隔超过`replica.lag.time.max.ms`。Kafka会认为此replica已经死亡会从同步副本列表从移除。检测慢副本机制发生了变化——如果一个replica开始落后leader超过`replica.lag.time.max.ms`。Kafka会认为太缓慢并且会从同步副本列表中移除。除非replica请求leader时间间隔大于`replica.lag.time.max.ms`，因此即使leader使流量激增和大批量写消息。Kafka也不会从同步副本列表从移除该副本。

7.5 一致性保证

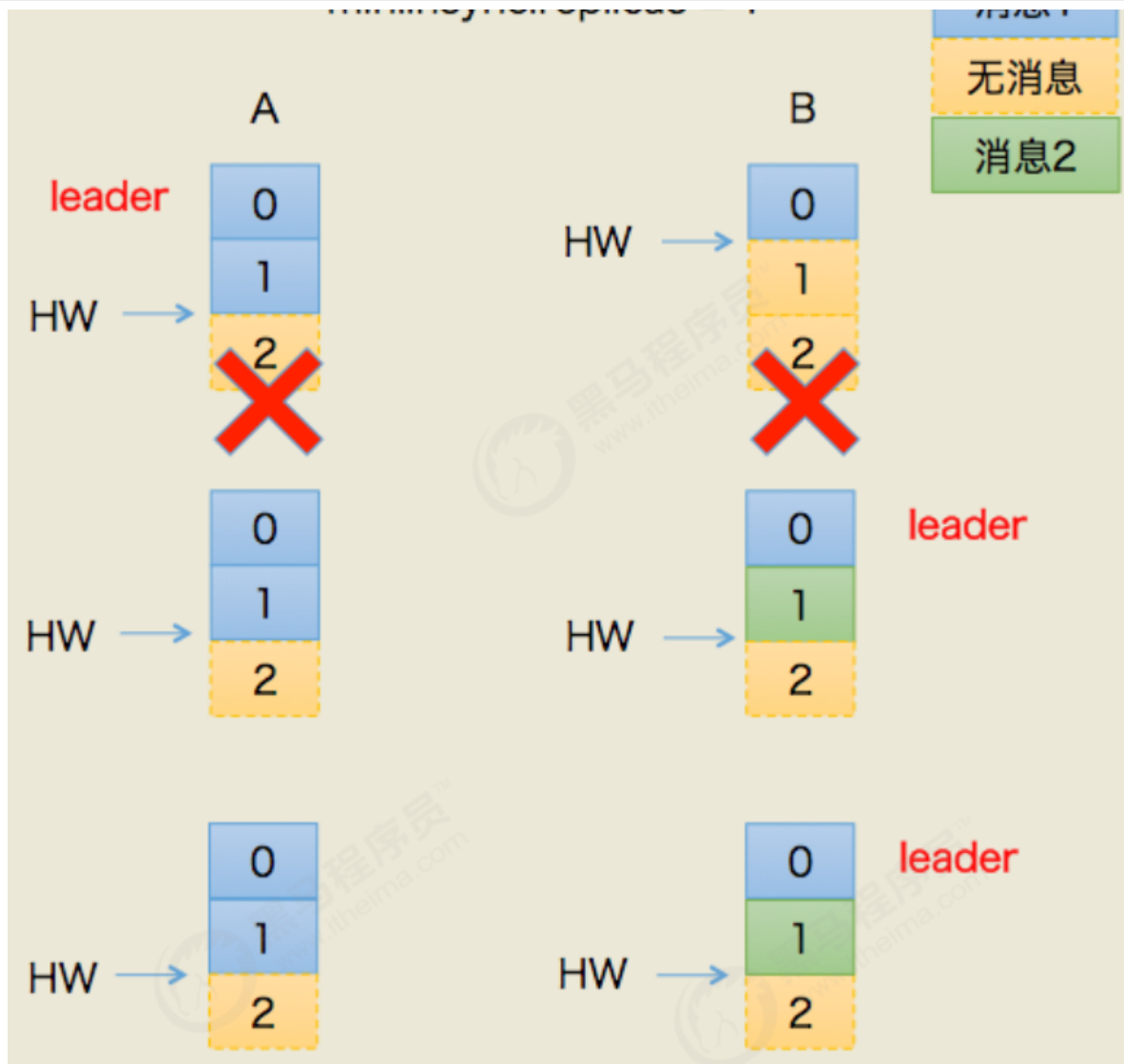
- 在leader宕机后，只能从ISR列表选取新的leader，无论ISR中哪个副本被选为新的leader，它都知道HW之前的数据，可以保证在切换了leader后，消费者可以继续看到HW之前已经提交的数据。
- HW的截断机制：选出了新的leader，而新的leader并不能保证已经完全同步了之前leader的所有数据，只能保证HW之前的数据是同步过的，此时所有的follower都要将数据截断到HW的位置，再和新的leader同步数据，来保证数据一致。当宕机的leader恢复，发现新的leader中的数据和自己持有的数据不一致，此时宕机的leader会将自己的数据截断到宕机之前的hw位置，然后同步新leader的数据。宕机的leader活过来也像follower一样同步数据，来保证数据的一致性。

Leader Epoch引用

数据丢失场景



数据出现不一致场景



Kafka 0.11.0.0版本解决方案

造成上述两个问题的根本原因在于HW值被用于衡量副本备份的成功与否以及在出现failure时作为日志截断的依据，但HW值的更新是异步延迟的，特别是需要额外的FETCH请求处理流程才能更新，故这中间发生的任何崩溃都可能导致HW值的过期。鉴于这些原因，Kafka 0.11引入了leader epoch来取代HW值。Leader端多开辟一段内存区域专门保存leader的epoch信息，这样即使出现上面的两个场景也能很好地规避这些问题。

所谓leader epoch实际上是一对值：(epoch, offset)。epoch表示leader的版本号，从0开始，当leader变更过1次时epoch就会+1，而offset则对应于该epoch版本的leader写入第一条消息的位移。因此假设有两对值：

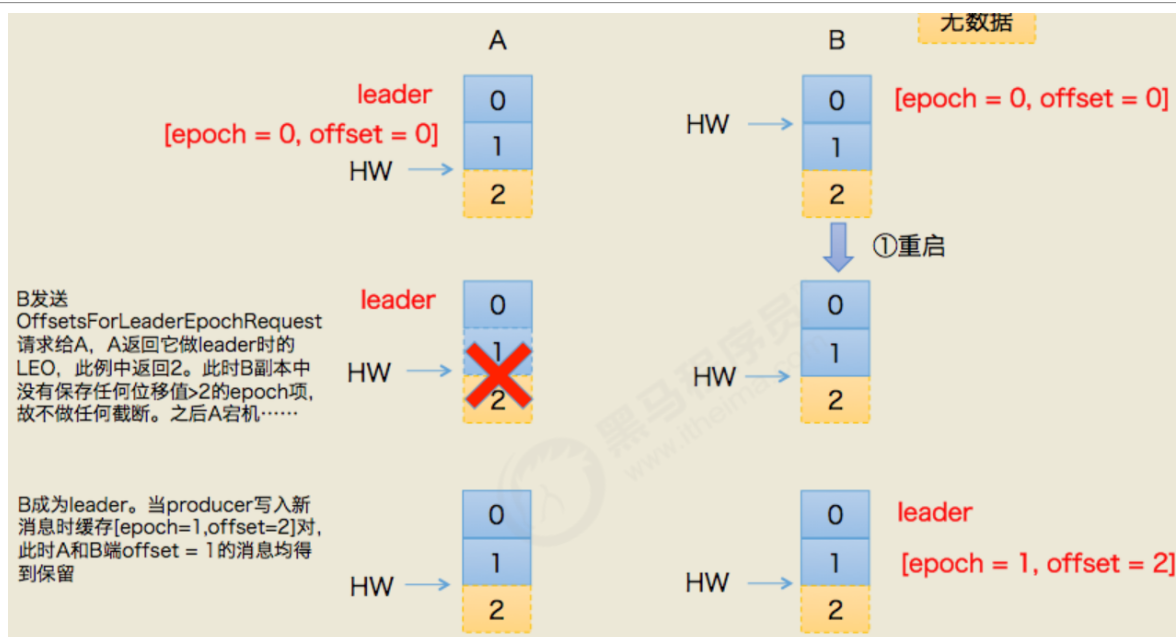
(0, 0)

(1, 120)

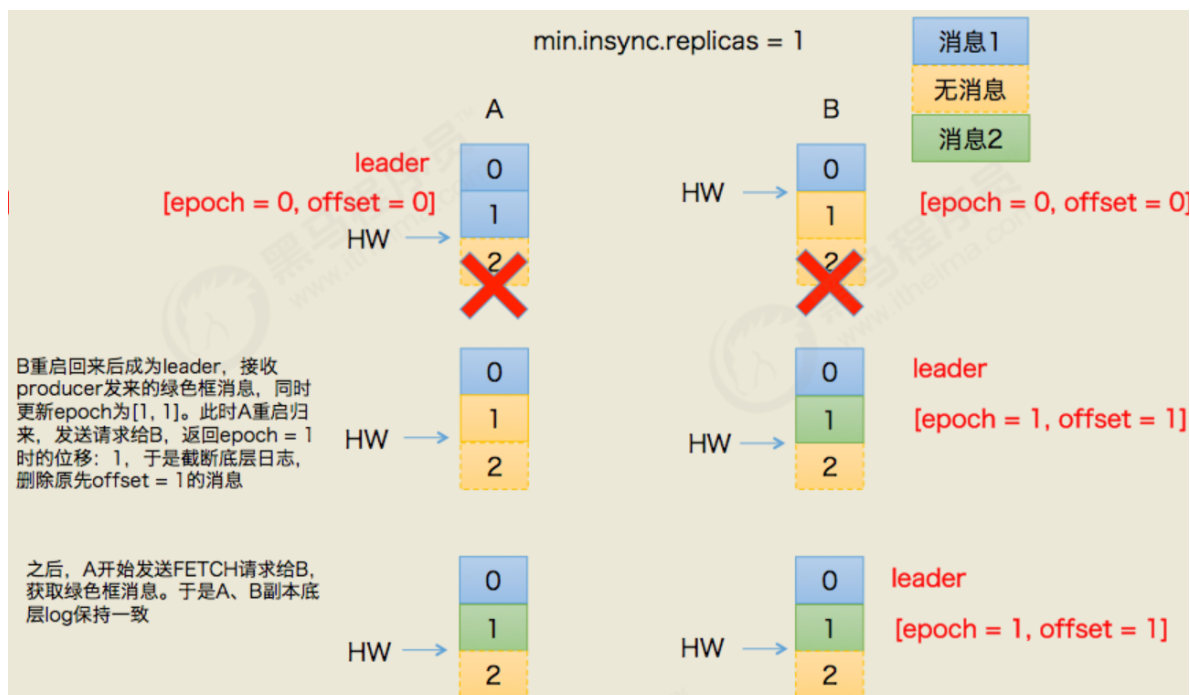
则表示第一个leader从位移0开始写入消息；共写了120条[0, 119]；而第二个leader版本号是1，从位移120处开始写入消息。

leader broker中会保存这样的一个缓存，并定期地写入到一个checkpoint文件中。

避免数据丢失：



避免数据不一致



7.6 消息重复的场景及解决方案

7.6.1 生产者端重复

生产发送的消息没有收到正确的broke响应, 导致producer重试。

producer发出一条消息, broke落盘以后因为网络等种种原因发送端得到一个发送失败的响应或者网络中断, 然后producer收到一个可恢复的Exception重试消息导致消息重复。

解决方案:

1、启动kafka的幂等性

要启动kafka的幂等性, 无需修改代码, 默认为关闭, 需要修改配置文件:enable.idempotence=true 同时要求 ack=all 且 retries>1。

2、ack=0, 不重试。

7.6.2 消费者端重复

1、根本原因

数据消费完没有及时提交offset到broker。

解决方案

1、取消自动提交

每次消费完或者程序退出时手动提交。这可能也没法保证一条重复。

2、下游做幂等

一般的解决方案是让下游做幂等或者尽量每消费一条消息都记录offset，对于少数严格的场景可能需要把offset或唯一ID,例如订单ID和下游状态更新放在同一个数据库里面做事务来保证精确的一次更新或者在下游数据表里面同时记录消费offset，然后更新下游数据的时候用消费位点做乐观锁拒绝掉旧位点的数据更新。

7.7 __consumer_offsets

__consumer_offsets是一个内部topic，对用户而言是透明的，除了它的数据文件以及偶尔在日志中出现这两点之外，用户一般是感觉不到这个topic的。不过我们的确知道它保存的是Kafka新版本consumer的位移信息。

7.7.1 何时创建

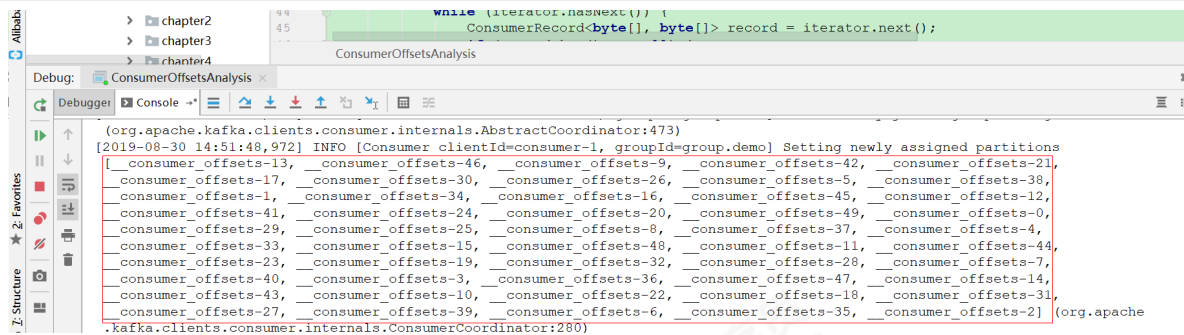
一般情况下，当集群中第一有消费者消费消息时会自动创建主题__consumer_offsets，分区数可以通过offsets.topic.num.partitions参数设定，默认值为50，如下

```
itcast@Server-node:/mnt/d/kafka_2.12-2.2.1$ ls /tmp/kafka/log/
consumer_offsets-0  consumer_offsets-30  consumer_offsets-8  transaction_state-29  transaction_state-6
consumer_offsets-1  consumer_offsets-31  consumer_offsets-9  transaction_state-3  transaction_state-7
consumer_offsets-10  consumer_offsets-32  transaction_state-0  transaction_state-30  transaction_state-8
consumer_offsets-11  consumer_offsets-33  transaction_state-1  transaction_state-31  transaction_state-9
consumer_offsets-12  consumer_offsets-34  transaction_state-10  transaction_state-32  schemas-0
consumer_offsets-13  consumer_offsets-35  transaction_state-11  transaction_state-33  cleaner-offset-checkpoint
consumer_offsets-14  consumer_offsets-36  transaction_state-12  transaction_state-34  heima-0
consumer_offsets-15  consumer_offsets-37  transaction_state-13  transaction_state-35  heima-1
consumer_offsets-16  consumer_offsets-38  transaction_state-14  transaction_state-36  heima-2
consumer_offsets-17  consumer_offsets-39  transaction_state-15  transaction_state-37  heima-par-0
consumer_offsets-18  consumer_offsets-4  transaction_state-16  transaction_state-38  heima-par-1
consumer_offsets-19  consumer_offsets-40  transaction_state-17  transaction_state-39  heima-par-2
consumer_offsets-2  consumer_offsets-41  transaction_state-18  transaction_state-4  heima-par-3
consumer_offsets-20  consumer_offsets-42  transaction_state-19  transaction_state-40  log-start-offset-checkpoint
consumer_offsets-21  consumer_offsets-43  transaction_state-2  transaction_state-41  meta.properties
consumer_offsets-22  consumer_offsets-44  transaction_state-20  transaction_state-42  recovery-point-offset-checkpoint
consumer_offsets-23  consumer_offsets-45  transaction_state-21  transaction_state-43  replication-offset-checkpoint
consumer_offsets-24  consumer_offsets-46  transaction_state-22  transaction_state-44  topic0701-0
consumer_offsets-25  consumer_offsets-47  transaction_state-23  transaction_state-45  topic0703-0
consumer_offsets-26  consumer_offsets-48  transaction_state-24  transaction_state-46  topic0703-1
consumer_offsets-27  consumer_offsets-49  transaction_state-25  transaction_state-47  topic0703-2
consumer_offsets-28  consumer_offsets-5  transaction_state-26  transaction_state-48  topic0703kafka_source-0
consumer_offsets-29  consumer_offsets-6  transaction_state-27  transaction_state-49  topic0828-0
consumer_offsets-3  consumer_offsets-7  transaction_state-28  transaction_state-5  topic0828-1
```

7.7.2 解析分区

见代码库：com.heima.kafka.chapter7.ConsumerOffsetsAnalysis

获取所有分区



总结

本章主要讲解了Kafka相关稳定性的操作，包括幂等性、事务的处理，同时对可靠性保证与一致性保证做了讲解，讲解了消息重复以及解决方案。

第8章 高级应用

*tips 学完这一章你可以

作为运维人员掌握命令行工具

使用Connect进行流信息处理

掌握延迟消息、流式处理等

Kafka和SpringBoot整合

8.1 命令行工具

参考官网：<http://kafka.apache.org/22/documentation.html>

8.1.1 消费组管理

- 查看消费组

```
itcast@Server-node:/mnt/d/kafka_2.12-2.2.1$ bin/kafka-consumer-groups.sh --  
bootstrap-server localhost:9092 --list  
itcast@Server-node:/mnt/d/kafka_2.12-2.2.1$
```

从上面可以看出，没有输出任何消费组信息，接下来我们启动一个消费端，再次查询即有消费组信息



```

17 // 主题名称-之前已经创建
18 private static final String topic = "heima";
19 // 消费组
20 private static
21 final String groupId = "group.demo";
22
23 public static void main(String[] args) {
24     Properties properties = new Properties();
25     properties.put("key.deserializer",
ConsumerFastStart > brokerList

```

```

INFO [Consumer clientId=consumer-1, groupId=group.demo] Revoking previously assigned partitions []
s.consumer.internals.ConsumerCoordinator:462)
INFO [Consumer clientId=consumer-1, groupId=group.demo] (Re-)joining group (org.apache.kafka.clients
ractorCoordinator:509)
INFO [Consumer clientId=consumer-1, groupId=group.demo] Successfully joined group with generation 1

```

```

itcast@Server-node:/mnt/d/kafka_2.12-2.2.1$ bin/kafka-consumer-groups.sh --
bootstrap-server localhost:9092 --list
group.demo
itcast@Server-node:/mnt/d/kafka_2.12-2.2.1$

```

• 查看消费组详情

```

itcast@Server-node:/mnt/d/kafka_2.12-2.2.1$ bin/kafka-consumer-groups.sh --
bootstrap-server localhost:9092 --describe --group group.demo

```

TOPIC	PARTITION	CURRENT-OFFSET	LOG-END-OFFSET	LAG	CONSUMER-ID	HOST	CLIENT-ID
heima	0	0	0	0	consumer-1-38efa901-4917-4660-ab66-3e5b989cbac3	/127.0.0.1	consumer-1
heima	1	0	0	0	consumer-1-38efa901-4917-4660-ab66-3e5b989cbac3	/127.0.0.1	consumer-1
heima	2	0	0	0	consumer-1-38efa901-4917-4660-ab66-3e5b989cbac3	/127.0.0.1	consumer-1

```

itcast@Server-node:/mnt/d/kafka_2.12-2.2.1$

```

• 查看消费组当前的状态

```

itcast@Server-node:/mnt/d/kafka_2.12-2.2.1$ bin/kafka-consumer-groups.sh --
bootstrap-server localhost:9092 --describe --group group.demo --state

```

COORDINATOR (ID)	ASSIGNMENT-STRATEGY	STATE
#MEMBERS		
Server-node.localdomain:9092 (0) range		Stable
1		

```

itcast@Server-node:/mnt/d/kafka_2.12-2.2.1$

```

• 消费组内成员信息



```
bootstrap-server localhost:9092 --describe --group group.demo --members
```

CONSUMER-ID	HOST	CLIENT-ID
#PARTITIONS		
consumer-1-38efa901-4917-4660-ab66-3e5b989cbac3	/127.0.0.1	consumer-1
3		

```
itcast@Server-node:/mnt/d/kafka_2.12-2.2.1$
```

- 删除消费组，如果有消费者在使用则会失败

```
itcast@Server-node:/mnt/d/kafka_2.12-2.2.1$ bin/kafka-consumer-groups.sh --
bootstrap-server localhost:9092 --delete --group group.demo
Error: Deletion of some consumer groups failed:
* Group 'group.demo' could not be deleted due to:
java.util.concurrent.ExecutionException:
org.apache.kafka.common.errors.GroupNotEmptyException: The group is not empty.
itcast@Server-node:/mnt/d/kafka_2.12-2.2.1$
```

8.1.2 消费位移管理

重置消费位移，前提是没有消费者在消费，提示信息如下：

```
itcast@Server-node:/mnt/d/kafka_2.12-2.2.1$ bin/kafka-consumer-groups.sh --
bootstrap-server localhost:9092 --group g
roup.demo --all-topics --reset-offsets --to-earliest --execute
Error: Assignments can only be reset if the group 'group.demo' is inactive, but
the current state is Stable.
```

TOPIC	PARTITION	NEW-OFFSET
dayuan@MY-20190430BUDR:		

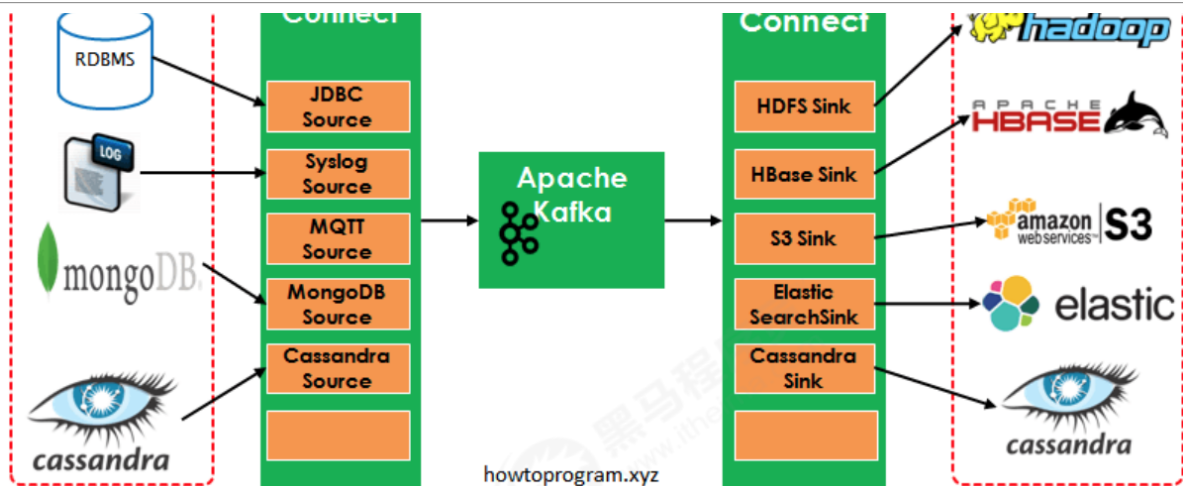
```
itcast@Server-node:/mnt/d/kafka_2.12-2.2.1$
```

参数：--all-topics指定了所有主题，可以修改为--topics，指定单个主题。

8.2 数据管道Connect

8.2.1 概述

Kafka是一个使用越来越广的消息系统，尤其是在大数据开发中（实时数据处理和分析）。为何集成其他系统和解耦应用，经常使用Producer来发送消息到Broker，并使用Consumer来消费Broker中的消息。Kafka Connect是到0.9版本才提供的并极大的简化了其他系统与Kafka的集成。Kafka Connect运用用户快速定义并实现各种Connector(File,Jdbc,Hdfs等)，这些功能让大批量数据导入/导出Kafka很方便。



在Kafka Connect中还有两个重要的概念：Task 和 Worker。

Connect中一些概念

连接器：实现了Connect API，决定需要运行多少个任务，按照任务来进行数据复制，从work进程获取任务配置并将其传递下去

任务：负责将数据移入或移出Kafka

work进程：相当与connector和任务的容器，用于负责管理连接器的配置、启动连接器和连接器任务，提供REST API

转换器：kafka connect和其他存储系统直接发送或者接受数据之间转换数据

8.2.2 独立模式--文件系统

场景

以下示例使用到了两个Connector，将文件source.txt 中的内容通过Source连接器写入Kafka主题中，然后将内容写入srouce.sink.txt中。

- FileStreamSource：从source.txt中读取并发布到Broker中
- FileStreamSink：从Broker中读取数据并写入到source.sink.txt文件中

步骤详情

首先我们来看下Worker进程用到的配置文件\${KAFKA_HOME}/config/connect-standalone.properties

```
// kafka集群连接的地址
bootstrap.servers=localhost:9092
// 格式转化类
key.converter=org.apache.kafka.connect.json.JsonConverter
value.converter=org.apache.kafka.connect.json.JsonConverter
// json消息中是否包含schema
key.converter.schemas.enable=true
value.converter.schemas.enable=true
// 保存偏移量的文件路径
offset.storage.file.filename=/tmp/connect.offsets
// 设定提交偏移量的频率
offset.flush.interval.ms=10000
```

其中的Source使用到的配置文件是\${KAFKA_HOME}/config/connect-file-source.properties



```
name=local-file-source
// 连接器的全限定名称，设置类名称也是可以的
connector.class=FileStreamSource
// task数量
tasks.max=1
// 数据源的文件路径
file=/tmp/source.txt
// 主题名称
topic=topic0703
```

其中的Sink使用到的配置文件是\${KAFKA_HOME}/config/connect-file-sink.properties

```
name=local-file-sink
connector.class=FileStreamSink
tasks.max=1
file=/tmp/source.sink.txt
topics=topic0703
```

启动source连接器

```
itcast@Server-node:/mnt/d/kafka_2.12-2.2.1$ bin/connect-standalone.sh
config/connect-standalone.properties config/connect-file-source.properties
```

启动sink连接器

```
itcast@Server-node:/mnt/d/kafka_2.12-2.2.1$ bin/connect-standalone.sh
config/connect-standalone.properties config/connect-file-sink.properties
```

source写入文本信息

```
itcast@Server-node:/mnt/d/kafka_2.12-2.2.1$ echo "Hello kafka,I
coming;">>/tmp/source.txt
```

查看sink文件

```
itcast@Server-node:/mnt/d/kafka_2.12-2.2.1$ cat /tmp/source.sink.txt
hello,kafka
I to do some
ello kafka,I coming;
Hello kafka,I coming;
```

8.2.3 信息流--ElasticSearch

概述

Kafka connect workers有两种工作模式，单机模式和分布式模式。在开发和适合使用单机模式的场景下，可以使用standalone模式，在实际生产环境下由于单个worker的数据压力会比较大，distributed模式对负载均衡和扩展性方面会有很大帮助。（本测试使用standalone模式）

关于Kafka Connect的详细情况可以参考[\[Kafka Connect\]](#)

Kafka Connect 安装

[\[Kafka Connect下载地址\]](#)



Worker配置

本测试使用standalone模式，因此修改../etc/schema-registry/connect-avro-standalone.properties

```
bootstrap.servers=localhost:9092
```

Elasticsearch Connector配置

修改../etc/kafka-connect-elasticsearch/quickstart-elasticsearch.properties

```
name=elasticsearch-sink
connector.class=io.confluent.connect.elasticsearch.ElasticsearchSinkConnector
tasks.max=1
//其中topics不仅对应Kafka的topic名称，同时也是Elasticsearch的索引名，
//当然也可以通过topic.index.map来设置从topic名到Elasticsearch索引名的映射
topics=topic0703
key.ignore=true
connection.url=http://localhost:9200
type.name=kafka-connect
```

启动

Elasticsearch

```
itcast@Server-node:~$ curl 'http://localhost:9200/?pretty'
{
  "name" : "MY-20190430BUDR",
  "cluster_name" : "elasticsearch",
  "cluster_uuid" : "ha3pnLkhRuGEIgXQstYnbQ",
  "version" : {
    "number" : "7.2.0",
    "build_flavor" : "default",
    "build_type" : "tar",
    "build_hash" : "508c38a",
    "build_date" : "2019-06-20T15:54:18.811730Z",
    "build_snapshot" : false,
    "lucene_version" : "8.0.0",
    "minimum_wire_compatibility_version" : "6.8.0",
    "minimum_index_compatibility_version" : "6.0.0-beta1"
  },
  "tagline" : "You know, for Search"
}
```

启动schema Registry

```
itcast@Server-node:/mnt/d/confluent-5.3.0$ bin/schema-registry-start etc/schema-registry/schema-registry.properties
[2019-07-22 06:01:00,059] INFO SchemaRegistryConfig values:
    access.control.allow.headers =
    access.control.allow.methods =
    access.control.allow.origin =
    authentication.method = NONE
    authentication.realm =
    authentication.roles = [*]
```

```
compression.enable = true
debug = false
host.name = MY-20190430BUDR.localdomain
idle.timeout.ms = 30000
inter.instance.headers.whitelist = []
inter.instance.protocol = http
kafkastore.bootstrap.servers = []
```

查看服务是否正常

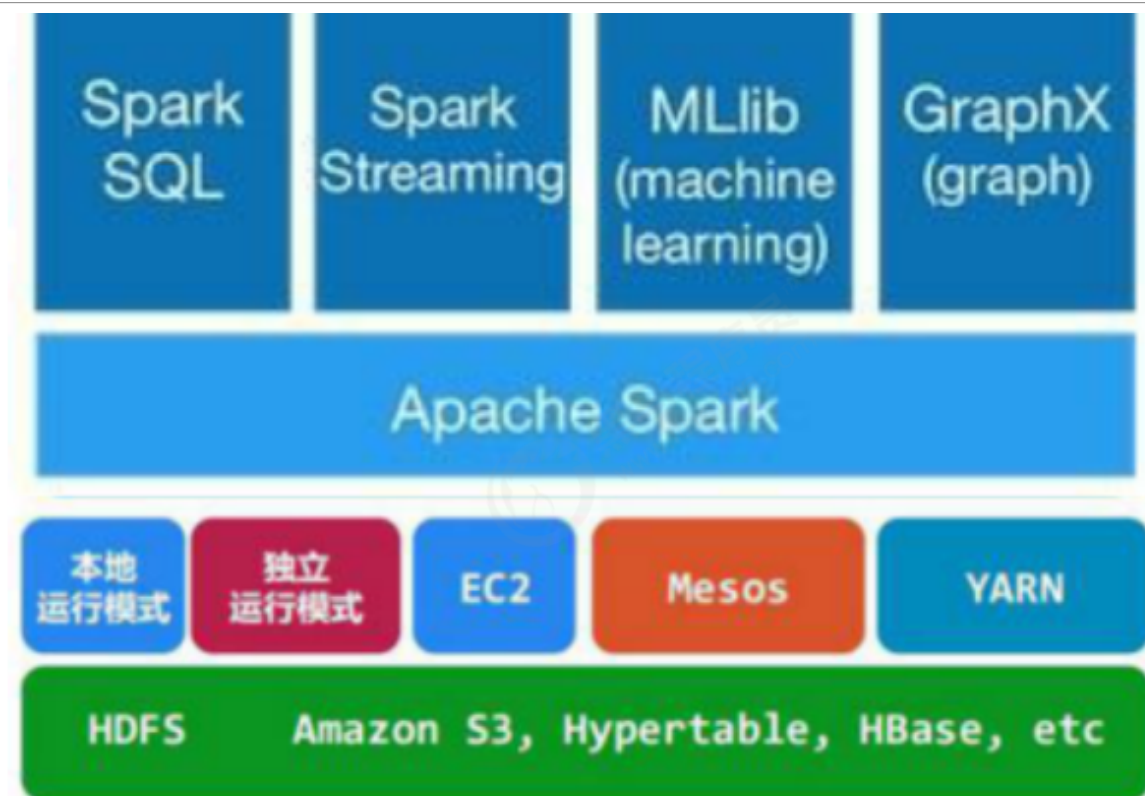
```
itcast@Server-node:~$ jps -l
1139 kafka.Kafka
2403 io.confluent.kafka.schemaregistry.rest.SchemaRegistryMain
2474 jdk.jcmd/sun.tools.jps.Jps
2172 org.elasticsearch.bootstrap.Elasticsearch
28 org.apache.zookeeper.server.quorum.QuorumPeerMain
```

启动Connector

```
itcast@Server-node:/mnt/d/confluent-5.3.0$ ./bin/connect-standalone etc/schema-
registry/connect-avro-standalone.properties etc/kafka-connect-
elasticsearch/quickstart-elasticsearch.properties
```

8.4 流式处理Spark

Spark最初诞生于美国加州大学伯克利分校（UC Berkeley）的AMP实验室，是一个可应用于大规模数据处理的快速、通用引擎。2013年，Spark加入Apache孵化器项目后，开始获得迅猛的发展，如今已成为Apache软件基金会最重要的三大分布式计算系统开源项目之一（即Hadoop、Spark、Storm）。Spark最初的设计目标是使数据分析更快——不仅运行速度快，也要能快速、容易地编写程序。为了使程序运行更快，Spark提供了内存计算，减少了迭代计算时的IO开销；而为了使编写程序更为容易，Spark使用简练、优雅的Scala语言编写，基于Scala提供了交互式的编程体验。虽然，Hadoop已成为大数据的事实标准，但其MapReduce分布式计算模型仍存在诸多缺陷，而Spark不仅具备Hadoop MapReduce所具有的优点，且解决了Hadoop MapReduce的缺陷。Spark正以其结构一体化、功能多元化的优势逐渐成为当今大数据领域最热门的大数据计算平台。



8.4.1 Spark安装与应用

官网

<http://spark.apache.org/downloads.html>

下载安装包解压即可

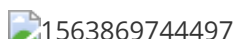
启动

```
itcast@Server-node:/mnt/d/spark-2.4.3-bin-hadoop2.7$ sbin/start-all.sh
starting org.apache.spark.deploy.master.Master, logging to /mnt/d/spark-2.4.3-
bin-hadoop2.7/logs/spark-dayuan-org.apache.spark.deploy.master.Master-1-MY-
20190430BUDR.out
itcast@localhost's password:
localhost: starting org.apache.spark.deploy.worker.Worker, logging to
/mnt/d/spark-2.4.3-bin-hadoop2.7/logs/spark-dayuan-
org.apache.spark.deploy.worker.worker-1-MY-20190430BUDR.out
```

验证

```
itcast@Server-node:/mnt/d/spark-2.4.3-bin-hadoop2.7$ jps -l
2819 kafka.Kafka
3972 jdk.jcmd/sun.tools.jps.Jps
3894 org.apache.spark.deploy.worker.Worker
28 org.apache.zookeeper.server.quorum.QuorumPeerMain
3726 org.apache.spark.deploy.master.Master
dayuan@MY-20190430BUDR:/mnt/d/spark-2.4.3-bin-hadoop2.7$
```

浏览器输入：<http://127.0.0.1:8080> 验证

1563869744497

 1563869975752

8.4.3 Spark和Kafka整合

见代码：com.spark.SparkStreamingFromkafka

演示

发送消息

```
itcast@Server-node:/mnt/d/kafka_2.12-2.2.1$ bin/kafka-console-producer.sh --
broker-list localhost:9092 --topic heima
>hello
>hello
>I coming;
>
>
>
>I coming;
>
```

接收消息

 1563874027371

 1563874206122

8.5 SpringBoot Kafka

kafka是一个消息队列产品，基于Topic partitions的设计，能达到非常高的消息发送处理性能。下面通过一个SpringBoot项目来演示整合Kafka。

8.5.1 创建SpringBoot项目

Kafka_Spring_Learn

验证主类：com.heima.kafka.KafkaApplication

8.5.2 Springboot整合Kafka

- 添加pom.xml


```
<!-- https://mvnrepository.com/artifact/org.springframework.kafka/spring-kafka -
-->
<dependency>
  <groupId>org.springframework.kafka</groupId>
  <artifactId>spring-kafka</artifactId>
  <version>2.2.6.RELEASE</version>
</dependency>
```

- 添加application.properties

Spring-kafka-producer: bootstrap.servers=127.0.0.1:9092

消息的发送

```
/**
 * 发送消息
 */
@GetMapping("/send/{input}")
public String sendToKafka(@PathVariable String input) {
    this.template.send(topic, input);
    return "send success";
}
```

 1564539421132

消息的接收

```
/**
 * 接收消息
 */
@KafkaListener(id = "", topics = topic, groupId = "group.demo")
public void listener(String input) {
    logger.info("input value: {}", input);
}
```

演示效果如下

```
2019-07-31 10:13:50.653 INFO 11228 --- [nio-8080-exec-3]
o.a.kafka.common.utils.AppInfoParser : Kafka version : 2.0.1
2019-07-31 10:13:50.653 INFO 11228 --- [nio-8080-exec-3]
o.a.kafka.common.utils.AppInfoParser : Kafka commitId : fa14705e51bd2ce5
2019-07-31 10:13:50.659 INFO 11228 --- [ad | producer-1]
org.apache.kafka.clients.Metadata : Cluster ID: VLMqkoXiTt26zv9Inatu2A
2019-07-31 10:13:50.718 INFO 11228 --- [ntainer#0-0-C-1]
com.dayuan.kafka.KafkaApplication : input value: kafka
```

8.5.3 事务操作

第一种方式

当输入参数为“error”值时，进行了回滚操作。



Spring Kafka producer transaction id prefix=KafkaTx

```
// 事务操作
template.executeInTransaction(t -> {
    t.send(topic, input);
    if ("error".equals(input)) {
        throw new RuntimeException("input is error");
    }
    t.send(topic, input + " author");
    return true;
});
return "send success";
```

第二种方式

```
@GetMapping("/sendt/{input}")
@Transactional(rollbackFor = RuntimeException.class)
public String sendToKafka2(@PathVariable String input) throws
    ExecutionException, InterruptedException {
    template.send(topic, input);
    if ("error".equals(input)) {
        throw new RuntimeException("input is error");
    }
    template.send(topic, input + " author");
    return "send success";
}
```

8.6 消息中间件选型对比

见附录一；

总结

本章主要针对的是Kafka的一些高级应用，作为运维人员经常使用的命令行工具，同时对Kafka的数据管道做了不同场景的演示，并且使用SpringBoot与Kafka做了整合。

第9章 集群管理

*tips 学完这一章你可以

熟悉Kafka集群管理相关内容

配置与调优

集群是一种计算机系统，它通过一组松散集成的计算机软件和/或硬件连接起来高度紧密地协作完成计算工作。在某种意义上，他们可以被看作是一台计算机。集群系统中的单个计算机通常称为节点，通常通过局域网连接，但也有其它的可能连接方式。集群计算机通常用来改进单个计算机的计算速度和/或可靠性。一般情况下集群计算机比单个计算机，比如工作站或超级计算机性能价格比要高得多。

北京市昌平区建材城西路金燕龙办公楼一层 电话：400-618-9090

集群拥有以下两个特点：

1. 可扩展性：集群的性能不限制于单一的服务实体，新的服务实体可以动态的添加到集群，从而增强集群的性能。
2. 高可用性：集群当其中一个节点发生故障时，这台节点上面所运行的应用程序将在另一台节点被自动接管，消除单点故障对于增强数据可用性、可达性和可靠性是非常重要的。

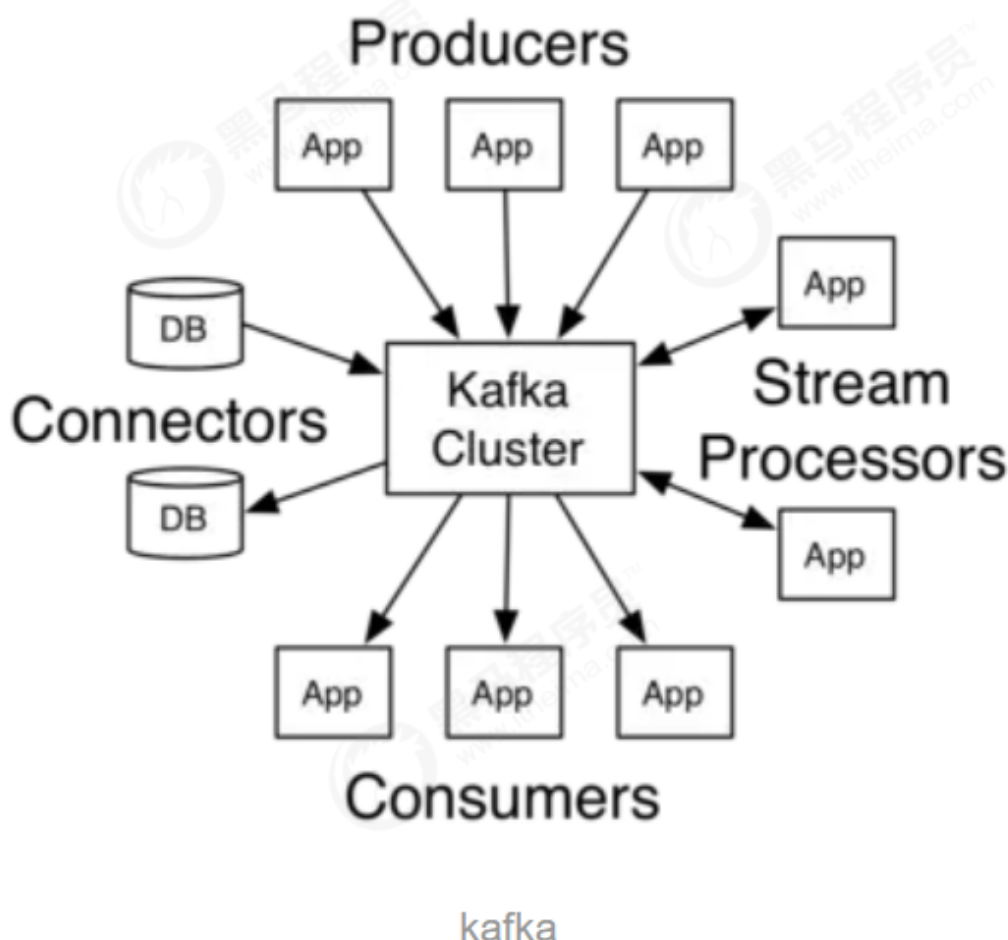
集群的能力

1. 负载均衡：负载均衡把任务比较均匀的分布到集群环境下的计算和网络资源，以提高数据吞吐量。
2. 错误恢复：如果集群中的某一台服务器由于故障或者维护需要无法使用，资源和应用程序将转移到可用的集群节点上。这种由于某个节点的资源不能工作，另一个可用节点中的资源能够透明的接管并继续完成任务的过程，叫做错误恢复。

负载均衡和错误恢复要求各服务实体中有执行同一任务的资源存在，而且对于同一任务的各个资源来说，执行任务所需的信息视图必须是相同的。

9.1 集群使用场景

Kafka 是一个分布式消息系统，具有高水平扩展和高吞吐量的特点。在Kafka 集群中，没有“中心主节点”的概念，集群中所有的节点都是对等的。

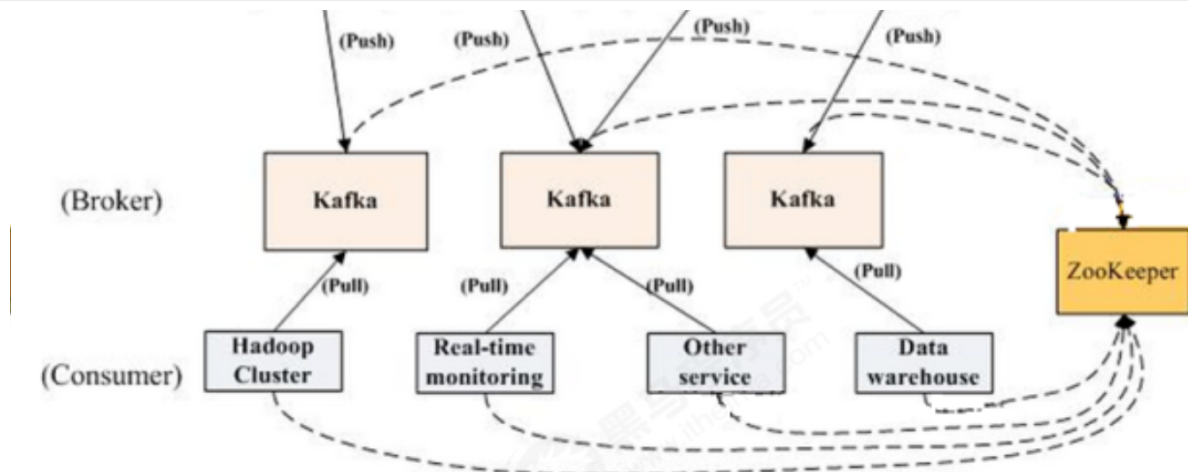


Broker（代理）

每个 Broker 即一个 Kafka 服务实例，多个 Broker 构成一个 Kafka 集群，生产者发布的消息将保存在 Broker 中，消费者将从 Broker 中拉取消息进行消费。

Kafka 集群架构图

北京市昌平区建材城西路金燕龙办公楼一层 电话：400-618-9090



从图中可以看出 Kafka 强依赖于 ZooKeeper，通过 ZooKeeper 管理自身集群，如：**Broker 列表管理、Partition 与 Broker 的关系、Partition 与 Consumer 的关系、Producer 与 Consumer 负载均衡、消费进度 Offset 记录、消费者注册**等，所以为了达到高可用，ZooKeeper 自身也必须是集群。

9.2 集群搭建

9.2.1 ZooKeeper集群搭建

场景

真实的集群是需要部署在不同的服务器上的，但是在我们测试时同时启动十几个虚拟机内存会吃不消，所以这里我们搭建**伪集群**，也就是把所有的服务都搭建在一台虚拟机上，用端口进行区分。

我们这里要求搭建一个三个节点的Zookeeper集群（伪集群）。

安装JDK

集群目录

创建zookeeper-cluster目录，将解压后的Zookeeper复制到以下三个目录

```
itcast@Server-node:/mnt/d/zookeeper-cluster$ ll
total 0
drwxrwxrwx 1 dayuan dayuan 512 Jul 24 10:02 ./
drwxrwxrwx 1 dayuan dayuan 512 Aug 19 18:42 ../
drwxrwxrwx 1 dayuan dayuan 512 Jul 24 10:02 zookeeper-1/
drwxrwxrwx 1 dayuan dayuan 512 Jul 24 10:02 zookeeper-2/
drwxrwxrwx 1 dayuan dayuan 512 Jul 24 10:02 zookeeper-3/
itcast@Server-node:/mnt/d/zookeeper-cluster$
```

ClientPort设置

配置每一个Zookeeper的dataDir（zoo.cfg）clientPort分别为2181 2182 2183

```
# the port at which the clients will connect
clientPort=2181
```

myid配置

在每个zookeeper的data目录下创建一个myid文件，内容分别是0、1、2。这个文件就是记录每个服务器的ID



```
temp/zookeeper/data/myid
0
dayuan@MY-20190430BUDR:/mnt/d/zookeeper-cluster/zookeeper-1$
```

zoo.cfg

在每一个zookeeper的 zoo.cfg配置客户端访问端口 (clientPort) 和集群服务器IP列表。

```
dayuan@MY-20190430BUDR:/mnt/d/zookeeper-cluster/zookeeper-1$ cat conf/zoo.cfg
# The number of milliseconds of each tick
# zk服务器的心跳时间
tickTime=2000
# The number of ticks that the initial
# synchronization phase can take
initLimit=10
# The number of ticks that can pass between
# sending a request and getting an acknowledgement
syncLimit=5
# the directory where the snapshot is stored.
# do not use /tmp for storage, /tmp here is just
# example sakes.
#dataDir=/tmp/zookeeper
dataDir=temp/zookeeper/data
dataLogDir=temp/zookeeper/log
# the port at which the clients will connect
clientPort=2181
# the maximum number of client connections.
# increase this if you need to handle more clients
#maxClientCnxns=60
#
# Be sure to read the maintenance section of the
# administrator guide before turning on autopurge.
#
# http://zookeeper.apache.org/doc/current/zookeeperAdmin.html#sc_maintenance
#
# The number of snapshots to retain in dataDir
#autopurge.snapRetainCount=3
# Purge task interval in hours
# Set to "0" to disable auto purge feature
#autopurge.purgeInterval=1

server.0=127.0.0.1:2888:3888
server.1=127.0.0.1:2889:3889
server.2=127.0.0.1:2890:3890
dayuan@MY-20190430BUDR:/mnt/d/zookeeper-cluster/zookeeper-1$
```

解释：server.服务器ID=服务器IP地址：服务器之间通信端口：服务器之间投票选举端口

启动集群

启动集群就是分别启动每个实例,启动后我们查询一下每个实例的运行状态

```
itcast@Server-node:/mnt/d/zookeeper-cluster/zookeeper-1$ bin/zkServer.sh status
Zookeeper JMX enabled by default
Using config: /mnt/d/zookeeper-cluster/zookeeper-1/bin/../conf/zoo.cfg
Mode: leader
北京市昌平区建材城西路金燕龙办公楼一层 电话：400-618-9090
```



```
Zookeeper JMX enabled by default
Using config: /mnt/d/zookeeper-cluster/zookeeper-2/bin/../conf/zoo.cfg
Mode: follower

itcast@Server-node:/mnt/d/zookeeper-cluster/zookeeper-3$ bin/zkServer.sh status
Zookeeper JMX enabled by default
Using config: /mnt/d/zookeeper-cluster/zookeeper-3/bin/../conf/zoo.cfg
Mode: follower
```

9.2.2 Kafka集群搭建

集群目录

```
itcast@Server-node:/mnt/d/kafka-cluster$ ll
total 0
drwxrwxrwx 1 dayuan dayuan 512 Aug 28 18:15 ./
drwxrwxrwx 1 dayuan dayuan 512 Aug 19 18:42 ../
drwxrwxrwx 1 dayuan dayuan 512 Aug 28 18:39 kafka-1/
drwxrwxrwx 1 dayuan dayuan 512 Jul 24 14:02 kafka-2/
drwxrwxrwx 1 dayuan dayuan 512 Jul 24 14:02 kafka-3/
drwxrwxrwx 1 dayuan dayuan 512 Aug 28 18:15 kafka-4/
itcast@Server-node:/mnt/d/kafka-cluster$
```

server.properties

```
# broker 编号，集群内必须唯一
broker.id=1
# host 地址
host.name=127.0.0.1
# 端口
port=9092
# 消息日志存放地址
log.dirs=/tmp/kafka/log/cluster/log3
# Zookeeper 地址，多个用,分隔
zookeeper.connect=localhost:2181,localhost:2182,localhost:2183
```

启动集群

分别通过 cmd 进入每个 Kafka 实例，输入命令启动



```
[2019-07-24 06:18:19,793] INFO [TransactionMarkerChannelManager 2]: Starting
(kafka.coordinator.transaction.TransactionMarkerChannelManager)
[2019-07-24 06:18:19,793] INFO [TransactionCoordinator id=2] Startup complete.
(kafka.coordinator.transaction.TransactionCoordinator)
[2019-07-24 06:18:19,846] INFO [/config/changes-event-process-thread]: Starting
(kafka.common.ZkNodeChangeNotificationListener$ChangeEventProcessThread)
[2019-07-24 06:18:19,869] INFO [SocketServer brokerId=2] started data-plane
processors for 1 acceptors (kafka.network.SocketServer)
[2019-07-24 06:18:19,879] INFO Kafka version: 2.2.1
(org.apache.kafka.common.utils.AppInfoParser)
[2019-07-24 06:18:19,879] INFO Kafka commitId: 55783d3133a5a49a
(org.apache.kafka.common.utils.AppInfoParser)
[2019-07-24 06:18:19,883] INFO [KafkaServer id=2] started
(kafka.server.KafkaServer)
```

9.3 多集群同步

MirrorMaker是为了解决Kafka跨集群同步、创建镜像集群而存在的；下图展示了其工作原理。该工具消费源集群消息然后将数据重新推送到目标集群。

 1563952426123

9.3.1 配置

创建镜像

使用MirrorMaker创建镜像是比较简单的，搭建好目标Kafka集群后，只需要启动mirror-maker程序即可。其中，一个或多个consumer配置文件、一个producer配置文件是必须的，whitelist、blacklist是可选的。在consumer的配置中指定源Kafka集群的Zookeeper，在producer的配置中指定目标集群的Zookeeper（或者broker.list）。

```
kafka-run-class.sh kafka.tools.MirrorMaker -
consumer.config sourceCluster1Consumer.config -
consumer.config sourceCluster2Consumer.config -num.streams 2 -
producer.config targetClusterProducer.config -whitelist=".*"
```

consumer配置文件：

```
# format: host1:port1,host2:port2 ...
bootstrap.servers=localhost:9092

# consumer group id
group.id=test-consumer-group

# what to do when there is no initial offset in kafka or if the current
# offset does not exist any more on the server: latest, earliest, none
#auto.offset.reset=
```

producer配置文件：



```
...kafka-server.properties...
```

```
# specify the compression codec for all data generated: none, gzip, snappy, lz4,
zstd
compression.type=none
```

9.3.2 调优

同步数据如何做到不丢失 首先发送到目标集群时需要确认：request.required.acks=1 发送时采用阻塞模式，否则缓冲区满了数据丢弃：queue.enqueue.timeout.ms=-1

总结

本章主要对Kafka集群展开讲解，介绍了集群使用场景，Zookeeper和Kafka多节点集群的搭建，以及多集群的同步操作。

第10章 监控

*tips 学完这一章你可以

知道Kafka的监控体系

掌握JMX监控指标

数据异动实时提醒

在开发工作当中，消费 Kafka 集群中的消息时，数据的变动是我们所关心的，当业务并不复杂的前提下，我们可以使用 Kafka 提供的命令工具，配合 Zookeeper 客户端工具，可以很方便的完成我们的工作。

10.1 监控度量指标

10.1.1 JMX

在实现Kafka监控系统的过程中，首先我们要知道监控的数据从哪来，Kafka自身提供的监控指标（包括broker和主题的指标，集群层面的指标通过各个broker的指标累加来获取）都可以通过JMX（Java Managent Extension）来进行获取。在使用JMX之前首先要确保Kafka开启了JMX的功能（默认是关闭的）

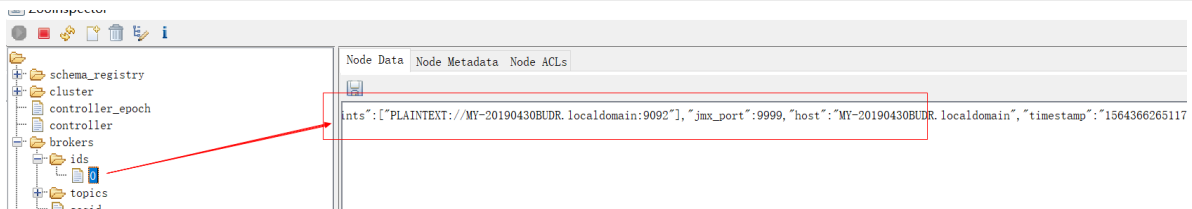
在kafka官网中 <http://kafka.apache.org/082/documentation.html#monitoring> 这样说：

Kafka uses Yammer Metrics for metrics reporting in both the server and the client. This can be configured to report stats using pluggable stats reporters to hook up to your monitoring system.

The easiest way to see the available metrics to fire up jconsole and point it at a running kafka client or server; this will all browsing all metrics with JMX.

在使用jmx之前需要确保kafka开启了jmx监控，kafka启动时要添加JMX_PORT=9999这一项，也就是：


```
itcast@Server-node:/mnt/d/kafka_2.12-2.2.1$ JMX_PORT=9999 bin/kafka-server-
start.sh config/server.properties
```




```
{"listener_security_protocol_map":{"PLAINTEXT":"PLAINTEXT"},  
"endpoints":["PLAINTEXT://Server-node.localdomain:9092"],  
"jmx_port":-1,"host":"Server-node.localdomain","timestamp":"1567042701998",  
"port":9092,"version":4}
```

10.1.2 JConsole

在开启JMX之后最简单的监控指标的方式就是使用JConsole，可以通过jconsole连接
service:jmx:rmi:///jndi/rmi://localhost:9999/jmxrmi或者localhost:9999来查看相应的数据值。

 1564369579315

 1564369784840

10.1.3 编程手段来获取监控指标

详情查看代码：com.heima.kafka.chapter10.JmxConnectionDemo

```
/**  
 * JMX Connection  
 */  
public class JmxConnectionDemo {  
    private MBeanServerConnection conn;  
    private String jmxURL;  
    private String ipAndPort;  
  
    public JmxConnectionDemo(String ipAndPort) {  
        this.ipAndPort = ipAndPort;  
    }  
  
    public boolean init(){  
        jmxURL = "service:jmx:rmi:///jndi/rmi://" + ipAndPort + "/jmxrmi";  
        try {  
            JMXServiceURL serviceURL = new JMXServiceURL(jmxURL);  
            JMXConnector connector = JMXConnectorFactory  
                .connect(serviceURL, null);  
            conn = connector.getMBeanServerConnection();  
            if (conn == null) {  
                return false;  
            }  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
        return true;  
    }  
  
    public double getMsgInPerSec() {
```



```
Object val = getAttribute(objectName, "OneMinuteRate");
if (val != null) {
    return (double) (Double) val;
}
return 0.0;
}

private Object getAttribute(String objName, String objAttr) {
    ObjectName objectName;
    try {
        objectName = new ObjectName(objName);
        return conn.getAttribute(objectName, objAttr);
    } catch (MalformedObjectNameException | IOException |
            ReflectionException | InstanceNotFoundException |
            AttributeNotFoundException | MBeanException e) {
        e.printStackTrace();
    }
    return null;
}

public static void main(String[] args) {
    JmxConnectionDemo jmxConnectionDemo =
        new JmxConnectionDemo("localhost:9999");
    jmxConnectionDemo.init();
    System.out.println(jmxConnectionDemo.getMsgInPerSec());
}
}
```

10.2 broker监控指标

10.2.1 活跃控制器

该指标表示 broker 是否就是当前的集群控制器，其值可以是 0 或 1。如果是 1，表示 broker 就是当前的控制器。任何时候，都应该只有一个 broker 是控制器，而且这个 broker 必须一直是集群控制器。如果出现了两个控制器，说明有一个本该退出的控制器线程被阻塞了，这会导致管理任务无陆正常执行，比如移动分区。为了解决这个问题，需要将这两个 broker 重启，而且不能通过正常的方式重启，因为此时它们无陆被正常关闭。

```
kafka.controller:type=KafkaController,name=ActiveControllerCount
```

值区间：0或1

10.2.2 请求处理器空闲率

Kafka 使用了两个线程来处理客户端的请求：网络处理器线程池和请求处理器线程池。网络处理器线程池负责通过网络读入和写出数据。这里没有太多的工作要做，也就是说，不用太过担心这些线程会出现问题。请求处理器线程池负责处理来自客户端的请求，包括从磁盘读取消息和往磁盘写入消息。因此，broker 负载的增长对这个线程池有很大的影响。

```
kafka.server:type=KafkaRequestHandlerPool,name=RequestHandlerAvgIdlePercent
```

10.2.3 主题流入字节

北京市昌平区建材城西路金燕龙办公楼一层 电话：400-618-9090



用于评估一个 broker 是否比集群里的其他 broker 接收了更多的流量，如果出现了这种情况，就需要对分区进行再均衡。

```
kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec
```

RateUnit 这是速率的时间段，在这里是“秒”。这两个属性表明，速率是通过 bis 来表示的，不管它的值是基于多长的时间段算出的平均值。速率还有其他 4 个不同粒度的属性。OneMinuteRate 前 1 分钟的平均值。FiveMinuteRate 前 5 分钟的平均值。FifteenMinuteRate 前 15 分钟的平均值。MeanRate 从 broker 启动到目前为止的平均值。

10.2.4 主题流出字节

主题流出字节速率与流入字节速率类似，是另一个与规模增长有关的度量指标。流出字节速率显示的是消费者从 broker 读取消息的速率。流出速率与流入速率的伸缩方式是不一样的，这要归功于 Kafka 对多消费者客户端的支持。

```
kafka.server:type=BrokerTopicMetrics,name=BytesOutPerSec
```

10.2.5 主题流入的消息

之前介绍的字节速率以字节的方式来表示 broker 的流量，而消息速率则以每秒生成消息个数的方式来表示流量，而且不考虑消息的大小。这也是一个很有用的生产者流量增长规模度量指标。它也可以与字节速率一起用于计算消息的平均大小。

```
kafka.server:type=BrokerTopicMetrics,name=MessagesInPerSec
```

10.2.6 分区数量

broker 的分区数量一般不会经常发生改变，它是指分配给 broker 的分区总数。它包括 broker 的每一个分区副本，不管是首领还是跟随者。

```
kafka.server:type=ReplicaManager,name=PartitionCount
```

10.2.7 首领数量

该度量指标表示 broker 拥有的首领分区数量。与 broker 的其他度量一样，该度量指标也应该在整个集群的 broker 上保持均等。我们需要对该指标进行周期性地检查，并适时地发出告警，即使在副本的数量和大小看起来都很完美的时候，它仍然能够显示出集群的不均衡问题。因为 broker 有可能出于各种原因释放掉一个分区的首领身份，比如 Zookeeper 会话过期，而在会话恢复之后，这个分区并不会自动拿回首领身份（除非启用了自动首领再均衡功能）。在这些情况下，该度量指标会显示较少的首领分区数，或者直接显示为零。这个时候需要运行一个默认的副本选举，重新均衡集群的首领。

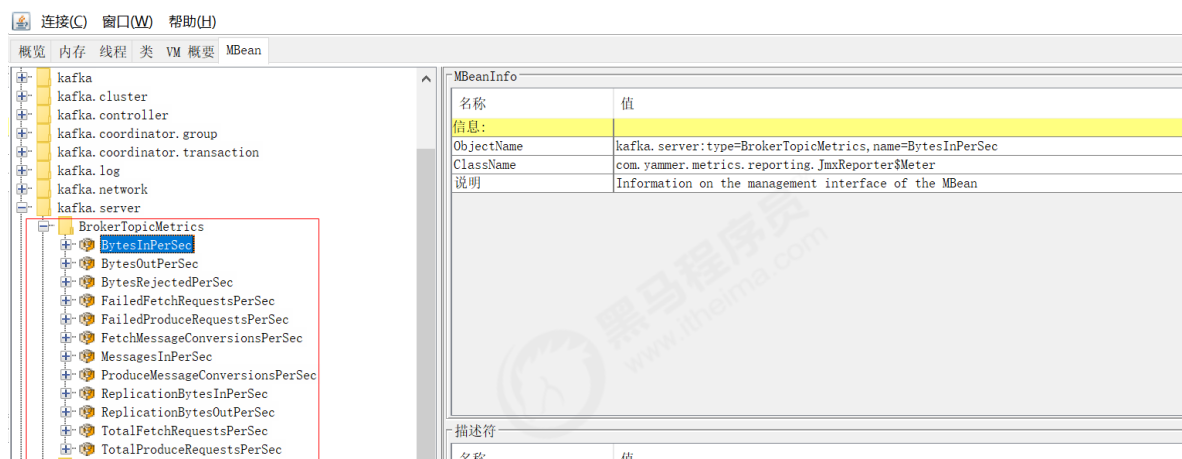
```
kafka.server:type=ReplicaManager,name=LeaderCount
```

10.3 主题分区监控

broker 的度量指标描述了 broker 的一般行为，除此之外，还有很多主题实例和分区实例的度量指标。

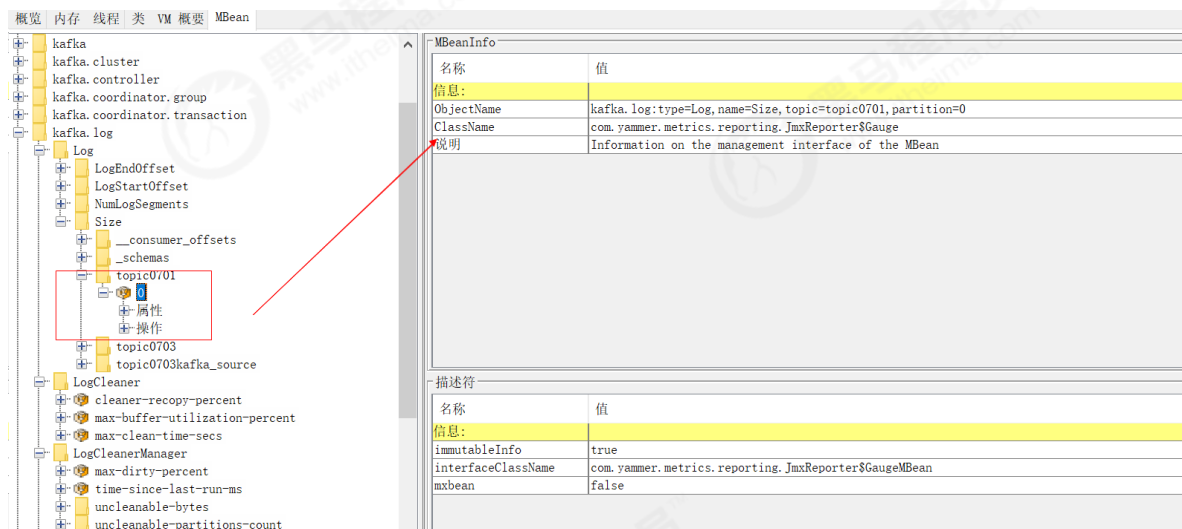
10.3.1 主题实例的度量指标

主题的数量，而且用户极有可能不会监控这些度量指标或设置告警。它们一般提供给客户端使用，客户端依此评估它们对 Kafka 的使用情况，并进行问题调试。



10.3.2 分区实例的度量指标

分区实例的度量指标不如主题实例的度量指标那样有用。另外，它们的数量会更加庞大，因为几百个主题就可能包含数千个分区。不过不管怎样，在某些情况下，它们还是有一定用处的。Partition size 度量指标表示分区当前在磁盘上保留的数据量。如果把它们组合在一起，就可以表示单个主题保留的数据量，作为客户端配额的依据。同一个主题的两个不同分区之间的数据量如果存在差异，说明消息并没有按照生产消息的键进行均匀分布。Log segment count 指标表示保存在磁盘上的日志片段的文件数量，可以与 Partition size 指标结合起来，用于跟踪资源的使用情况。



10.4 生产者监控指标

新版本 Kafka 生产者客户端的度量指标经过调整变得更加简洁，只用了少量的 MBean。相反，之前版本的客户端（不再受支持的版本）使用了大量的 MBean，而且度量指标包含了大量的细节（提供了大量的百分位和各种移动平均数）。这些度量指标提供了很大的覆盖面，但这样会让跟踪异常情况变得更加困难。生产者度量指标的 MBean 名字里都包含了生产者的客户端 ID。在下面的示例里，客户端 ID 使用 CLIENTID 表示，broker ID 使用 BROKERID 表示，主题的名字使用 TOPICNAME 表示，

```
kafka.server:type=BrokerTopicMetrics,name=ProduceMessageConversionsPerSec
kafka.server:type=BrokerTopicMetrics,name=TotalProduceRequestsPerSec
```

10.5 消费者监控指标

```
kafka.consumer:type=consumer-metrics,client-id=CLIENTID
kafka.consumer:type=consumer-fetch-manager-metrics,client-id=CLIENTID
```

10.6 Kafka Eagle

在开发工作当中，消费 Kafka 集群中的消息时，数据的变动是我们所关心的，当业务并不复杂的前提下，我们可以使用 Kafka 提供的命令工具，配合 Zookeeper 客户端工具，可以很方便的完成我们的工作。随着业务的复杂化，Group 和 Topic 的增加，此时我们使用 Kafka 提供的命令工具，已预感到力不从心，这时候 Kafka 的监控系统此刻便尤为显得重要，我们需要观察消费应用的详情。监控系统业界有很多杰出的开源监控系统。我们在早期，有使用 KafkaMonitor 和 Kafka Manager 等，不过随着业务的快速发展，以及互联网公司特有的一些需求，现有的开源的监控系统在性能、扩展性、和 DEVS 的使用效率方面，已经无法满足。因此，我们在过去的时间内，从互联网公司的一些需求出发，从各位 DEVS 的使用经验和反馈出发，结合业界的一些开源的 Kafka 消息监控，用监控的一些思考出发，设计开发了现在 Kafka 集群消息监控系统：Kafka Eagle。

Kafka Eagle 用于监控 Kafka 集群中 Topic 被消费的情况。包含 Lag 的产生，Offset 的变动，Partition 的分布，Owner，Topic 被创建的时间和修改的时间等信息。

安装下载

- [[Kafka Eagle 下载地址](#)]
- [[Kafka Eagle Github](#)]
- [Kafka Eagle 使用文档](#)

配置

D:\kafka-eagle-web-1.3.6\conf\system-config.properties

```
#####
# multi zookeeper&kafka cluster list
#####
#如果只有一个集群的话，就写一个cluster1就行了
kafka.eagle.zk.cluster.alias=cluster1
#这里填上刚才上准备工作中的zookeeper.connect地址
cluster1.zk.list=localhost:2181
#如果多个集群，继续写，如果没有注释掉
#cluster2.zk.list=xdn10:2181,xdn11:2181,xdn12:2181

#####
# zk client thread limit
#####
kafka.zk.limit.size=25

#####
# kafka eagle webui port
#####
###web界面地址端口
kafka.eagle.webui.port=8048

#####
# kafka offset storage
#####
cluster1.kafka.eagle.offset.storage=kafka
```



```
#####
# enable kafka metrics
#####
kafka.eagle.metrics.charts=false
kafka.eagle.sql.fix.error=false

#####
# kafka sql topic records max
#####
kafka.eagle.sql.topic.records.max=5000

#####
# alarm email configure
#####
#kafka.eagle.mail.enable=false
#kafka.eagle.mail.sa=alert_sa@163.com
#kafka.eagle.mail.username=alert_sa@163.com
#kafka.eagle.mail.password=mqslimczkdqabbbh
#kafka.eagle.mail.server.host=smtp.163.com
#kafka.eagle.mail.server.port=25

#####
# alarm im configure
#####
#kafka.eagle.im.dingding.enable=true
#kafka.eagle.im.dingding.url=https://oapi.dingtalk.com/robot/send?access_token=

#kafka.eagle.im.wechat.enable=true
#kafka.eagle.im.wechat.token=https://qyapi.weixin.qq.com/cgi-bin/gettoken?
corpid=xxx&corpsecret=xxx
#kafka.eagle.im.wechat.url=https://qyapi.weixin.qq.com/cgi-bin/message/send?
access_token=
#kafka.eagle.im.wechat.touser=
#kafka.eagle.im.wechat.toparty=
#kafka.eagle.im.wechat.totag=
#kafka.eagle.im.wechat.agentid=

#####
# delete kafka topic token
#####
kafka.eagle.topic.token=keadmin

#####
# kafka sasl authenticate
#####
#cluster1.kafka.eagle.sasl.enable=false
#cluster1.kafka.eagle.sasl.protocol=SASL_PLAINTEXT
#cluster1.kafka.eagle.sasl.mechanism=PLAIN
#cluster2.kafka.eagle.sasl.enable=false
#cluster2.kafka.eagle.sasl.protocol=SASL_PLAINTEXT
#cluster2.kafka.eagle.sasl.mechanism=PLAIN

#cluster1.kafka.eagle.sasl.client=/mnt/d/kafka-eagle-web-
1.3.6/conf/kafka_client_jaas.conf
#####
# kafka jdbc driver address
```

```
#这个地址，按照安装目录进行配置
kafka.eagle.url=jdbc:sqlite:D:/kafka-eagle-web-1.3.6/db/ke.db
kafka.eagle.username=root
kafka.eagle.password=123456
```

环境变量：KE_HOME D:\kafka-eagle-web-1.3.6

启动

启动命令：D:\kafka-eagle-web-1.3.6\bin\ke.bat

访问

<http://localhost:8048/ke> 默认用户名：admin 密码：admin

1564630826737

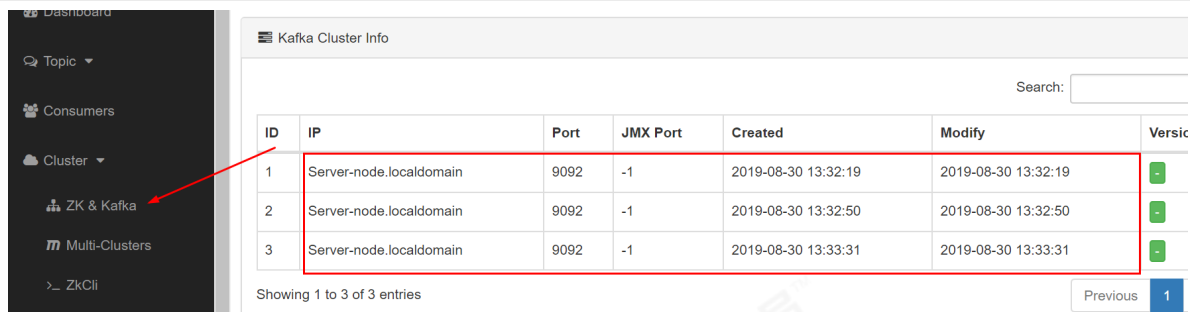
演示

ID	Topic Name	Partition Indexes	Partition Numbers	Created	Modify	Operate
1	topic0701	[0]	1	2019-07-09 17:21:44	2019-07-09 17:21:44	Remove
2	_schemas	[0]	1	2019-07-22 14:01:01	2019-07-22 14:01:01	Remove
3	__transaction_state	[44, 45, 4...	50	2019-07-31 14:15:21	2019-07-31 14:15:21	Remove
4	topic0828	[0, 1]	2	2019-08-28 09:35:24	2019-08-28 09:35:24	Remove
5	heima	[0, 1, 2]	3	2019-08-28 13:51:35	2019-08-28 16:42:51	Remove
6	heima-par	[0, 1, 2, 3]	4	2019-08-28 18:00:52	2019-08-30 09:00:03	Remove

- 启动一个消费端

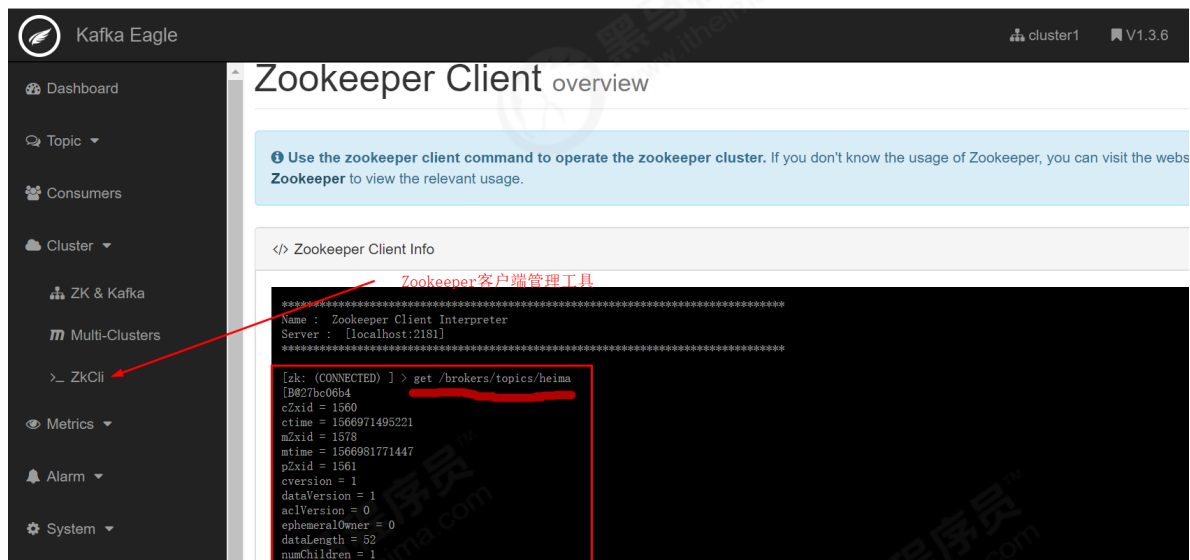
ID	Group	Topics	Node	Active Numbers
1	group.demo	1	Server-node.localdomain:9092	1

- 启动一个有三个节点的集群



ID	IP	Port	JMX Port	Created	Modify	Version
1	Server-node.localdomain	9092	-1	2019-08-30 13:32:19	2019-08-30 13:32:19	-
2	Server-node.localdomain	9092	-1	2019-08-30 13:32:50	2019-08-30 13:32:50	-
3	Server-node.localdomain	9092	-1	2019-08-30 13:33:31	2019-08-30 13:33:31	-

- Zookeeper客户端管理



Zookeeper Client overview

Use the zookeeper client command to operate the zookeeper cluster. If you don't know the usage of Zookeeper, you can visit the website [Zookeeper](#) to view the relevant usage.

Zookeeper Client Info

```
[zk: (CONNECTED) ] > get /brokers/topics/heim
[B027bc06b4
cZxid = 1560
ctime = 1566971495221
mZxid = 1578
mtime = 1566981771447
pZxid = 1561
cversion = 1
dataVersion = 1
aclVersion = 0
ephemeralOwner = 0
dataLength = 52
numChildren = 1
```

总结

本章主要讲解了Kafka的监控体系，了解度量指标，通过编程手段来获取监控指标，同时对生产者和消费者进行监控。同时引入第三方监控工具Eagle。

附录一：常用MQ对比

一、资料文档

Kafka：中。有kafka作者自己写的书，网上资料也有一些。

rabbitmq：多。有一些不错的书，网上资料多。

zeromq：少。没有专门写zeromq的书，网上的资料多是一些代码的实现和简单介绍。

rocketmq：少。没有专门写rocketmq的书，网上的资料良莠不齐，官方文档很简洁，但是对技术细节没有过多的描述。

activemq：多。没有专门写activemq的书，网上资料多。

二、开发语言

Kafka：Scala

rabbitmq：Erlang



rocketmq : java

activemq : java

三、支持的协议

Kafka : 自己定义的一套... (基于TCP)

rabbitmq : AMQP

zeromq : TCP、UDP

rocketmq : 自己定义的一套...

activemq : OpenWire、STOMP、REST、XMPP、AMQP

四、消息存储

Kafka : 内存、磁盘、数据库。支持大量堆积。

kafka的最小存储单元是分区，一个topic包含多个分区，kafka创建主题时，这些分区会被分配在多个服务器上，通常一个broker一台服务器。

分区首领会均匀地分布在不同的服务器上，分区副本也会均匀的分布在不同的服务器上，确保负载均衡和高可用性，当新的broker加入集群的时候，部分副本会被移动到新的broker上。

根据配置文件中的目录清单，kafka会把新的分区分配给目录清单里分区数最少的目录。

默认情况下，分区器使用轮询算法把消息均衡地分布在同一个主题的不同分区中，对于发送时指定了key的情况，会根据key的hashcode取模后的值存到对应的分区中。

rabbitmq : 内存、磁盘。支持少量堆积。

rabbitmq的消息分为持久化的消息和非持久化消息，不管是持久化的消息还是非持久化的消息都可以写入到磁盘。

持久化的消息在到达队列时就写入到磁盘，并且如果可以，持久化的消息也会在内存中保存一份备份，这样可以提高一定的性能，当内存吃紧的时候会从内存中清除。非持久化的消息一般只存在于内存中，在内存吃紧的时候会被换入到磁盘中，以节省内存。

引入镜像队列机制，可将重要队列“复制”到集群中的其他broker上，保证这些队列的消息不会丢失。配置镜像的队列，都包含一个主节点master和多个从节点slave,如果master失效，加入时间最长的slave会被提升为新的master，除发送消息外的所有动作都向master发送，然后由master将命令执行结果广播给各个slave，rabbitmq会让master均匀地分布在不同的服务器上，而同一个队列的slave也会均匀地分布在不同的服务器上，保证负载均衡和高可用性。

zeromq : 消息发送端的内存或者磁盘中。不支持持久化。

rocketmq : 磁盘。支持大量堆积。

commitLog文件存放实际的消息数据，每个commitLog上限是1G，满了之后会自动新建一个commitLog文件保存数据。ConsumeQueue队列只存放offset、size、tagcode，非常小，分布在多个broker上。ConsumeQueue相当于CommitLog的索引文件，消费者消费时会从consumeQueue中查找消息在commitLog中的offset，再去commitLog中查找元数据。

ConsumeQueue存储格式的特性，保证了写过程的顺序写盘（写CommitLog文件），大量数据IO都在顺序写同一个commitLog，满1G了再写新的。加上rocketmq是累计4K才强制从PageCache中刷到磁盘（缓存），所以高并发写性能突出。

activemq : 内存、磁盘、数据库。支持少量堆积。



Kafka：支持

rabbitmq：支持。

客户端将信道设置为事务模式，只有当消息被rabbitMq接收，事务才能提交成功，否则在捕获异常后进行回滚。使用事务会使得性能有所下降

zeromq：不支持

rocketmq：支持

activemq：支持

六、负载均衡

Kafka：支持负载均衡。

1>一个broker通常就是一台服务器节点。对于同一个Topic的不同分区，Kafka会尽力将这些分区分布到不同的Broker服务器上，zookeeper保存了broker、主题和分区的元数据信息。分区首领会处理来自客户端的生产请求，kafka分区首领会被分配到不同的broker服务器上，让不同的broker服务器共同分担任务。

每一个broker都缓存了元数据信息，客户端可以从任意一个broker获取元数据信息并缓存起来，根据元数据信息知道要往哪里发送请求。

2>kafka的消费者组订阅同一个topic，会尽可能地使得每一个消费者分配到相同数量的分区，分摊负载。

3>当消费者加入或者退出消费者组的时候，还会触发再均衡，为每一个消费者重新分配分区，分摊负载。

kafka的负载均衡大部分是自动完成的，分区的创建也是kafka完成的，隐藏了很多细节，避免了繁琐的配置和人为疏忽造成的负载问题。

4>发送端由topic和key来决定消息发往哪个分区，如果key为null，那么会使用轮询算法将消息均衡地发送到同一个topic的不同分区中。如果key不为null，那么会根据key的hashcode取模计算出要发往的分区。

rabbitmq：对负载均衡的支持不好。

1>消息被投递到哪个队列是由交换器和key决定的，交换器、路由键、队列都需要手动创建。

rabbitmq客户端发送消息要和broker建立连接，需要事先知道broker上有哪些交换器，有哪些队列。通常要声明要发送的目标队列，如果没有目标队列，会在broker上创建一个队列，如果有，就什么都不处理，接着往这个队列发送消息。假设大部分繁重任务的队列都创建在同一个broker上，那么这个broker的负载就会过大。（可以在上线前预先创建队列，无需声明要发送的队列，但是发送时不会尝试创建队列，可能出现找不到队列的问题，rabbitmq的备份交换器会把找不到队列的消息保存到一个专门的队列中，以便以后查询使用）

使用镜像队列机制建立rabbitmq集群可以解决这个问题，形成master-slave的架构，master节点会均匀分布在不同的服务器上，让每一台服务器分摊负载。slave节点只是负责转发，在master失效时会选择加入时间最长的slave成为master。

当新节点加入镜像队列的时候，队列中的消息不会同步到新的slave中，除非调用同步命令，但是调用命令后，队列会阻塞，不能在生产环境中调用同步命令。

2>当rabbitmq队列拥有多个消费者的时候，队列收到的消息将以轮询的分发方式发送给消费者。每条消息只会发送给订阅列表里的一个消费者，不会重复。

这种方式非常适合扩展，而且是专门为并发程序设计的。

3>对于rabbitmq而言，客户端与集群建立的TCP连接不是与集群中所有的节点建立连接，而是挑选其中一个节点建立连接。

但是rabbitmq集群可以借助HAProxy、LVS技术，或者在客户端使用算法实现负载均衡，引入负载均衡之后，各个客户端的连接可以分摊到集群的各个节点之中。

客户端均衡算法：

1)轮询法。按顺序返回下一个服务器的连接地址。

2)加权轮询法。给配置高、负载低的机器配置更高的权重，让其处理更多的请求；而配置低、负载高的机器，给其分配较低的权重，降低其系统负载。

3)随机法。随机选取一个服务器的连接地址。

4)加权随机法。按照概率随机选取连接地址。

5)源地址哈希法。通过哈希函数计算得到的一个数值，用该数值对服务器列表的大小进行取模运算。

6)最小连接数法。动态选择当前连接数最少的一台服务器的连接地址。

zeromq：去中心化，不支持负载均衡。本身只是一个多线程网络库。

rocketmq：支持负载均衡。

一个broker通常是一个服务器节点，broker分为master和slave,master和slave存储的数据一样，slave从master同步数据。

1>nameserver与每个集群成员保持心跳，保存着Topic-Broker路由信息，同一个topic的队列会分布在不同的服务器上。

2>发送消息通过轮询队列的方式发送，每个队列接收平均的消息量。发送消息指定topic、tags、keys，无法指定投递到哪个队列（没有意义，集群消费和广播消费跟消息存放在哪个队列没有关系）。

tags选填，类似于 Gmail 为每封邮件设置的标签，方便服务器过滤使用。目前只支持每个消息设置一个 tag，所以也可以类比为 Notify 的 MessageType 概念。

keys选填，代表这条消息的业务关键词，服务器会根据 keys 创建哈希索引，设置后，可以在 Console 系统根据 Topic、Keys 来查询消息，由于是哈希索引，请尽可能保证 key 唯一，例如订单号，商品 Id 等。

3>rocketmq的负载均衡策略规定：Consumer数量应该小于等于Queue数量，如果Consumer超过Queue数量，那么多余的Consumer 将不能消费消息。这一点和kafka是一致的，rocketmq会尽可能地为每一个Consumer分配相同数量的队列，分摊负载。

activemq：支持负载均衡。可以基于zookeeper实现负载均衡。

七、集群方式

Kafka：天然的'Leader-Slave'无状态集群，每台服务器既是Master也是Slave。

分区首领均匀地分布在不同的kafka服务器上，分区副本也均匀地分布在不同的kafka服务器上，所以每一台kafka服务器既含有分区首领，同时又含有分区副本，每一台kafka服务器是某一台kafka服务器的Slave，同时也是某一台kafka服务器的leader。

kafka的集群依赖于zookeeper，zookeeper支持热扩展，所有的broker、消费者、分区都可以动态加入移除，而无需关闭服务，与不依靠zookeeper集群的mq相比，这是最大的优势。

rabbitmq：支持简单集群，'复制'模式，对高级集群模式支持不好。

在rabbitmq集群中创建队列，集群只会在单个节点创建队列进程和完整的队列信息（元数据、状态、内容），而不是在所有节点上创建。

引入镜像队列，可以避免单点故障，确保服务的可用性，但是需要人为地为某些重要的队列配置镜像。

zeromq：去中心化，不支持集群。

rocketmq：常用 多对'Master-Slave' 模式，开源版本需手动切换Slave变成Master

Name Server是一个几乎无状态节点，可集群部署，节点之间无任何信息同步。

Broker部署相对复杂，Broker分为Master与Slave，一个Master可以对应多个Slave，但是一个Slave只能对应一个Master，Master与Slave的对应关系通过指定相同的BrokerName，不同的BrokerId来定义，BrokerId为0表示Master，非0表示Slave。Master也可以部署多个。每个Broker与Name Server集群中的所有节点建立长连接，定时注册Topic信息到所有Name Server。

Producer与Name Server集群中的其中一个节点（随机选择）建立长连接，定期从Name Server取Topic路由信息，并向提供Topic服务的Master建立长连接，且定时向Master发送心跳。Producer完全无状态，可集群部署。

Consumer与Name Server集群中的其中一个节点（随机选择）建立长连接，定期从Name Server取Topic路由信息，并向提供Topic服务的Master、Slave建立长连接，且定时向Master、Slave发送心跳。Consumer既可以从Master订阅消息，也可以从Slave订阅消息，订阅规则由Broker配置决定。

客户端先找到NameServer，然后通过NameServer再找到 Broker。

一个topic有多个队列，这些队列会均匀地分布在不同的broker服务器上。rocketmq队列的概念和kafka的分区概念是基本一致的，kafka同一个topic的分区尽可能地分布在不同的broker上，分区副本也会分布在不同的broker上。

rocketmq集群的slave会从master拉取数据备份，master分布在不同的broker上。

activemq：支持简单集群模式，比如'主-备'，对高级集群模式支持不好。

八、管理界面

Kafka：一般

rabbitmq：好

zeromq：无

rocketmq：无

activemq：一般

九、可用性

Kafka：非常高（分布式）

rabbitmq：高（主从）

zeromq：高。

rocketmq：非常高（分布式）

activemq：高（主从）

十、消息重复

Kafka：支持at least once、at most once



zeromq：只有重传机制，但是没有持久化，消息去了重传也没有用。既不是at least once、也不是at most once、更不是exactly only once

rocketmq：支持at least once

activemq：支持at least once

十一、吞吐量TPS

Kafka：极大

Kafka按批次发送消息和消费消息。发送端将多个小消息合并，批量发向Broker，消费端每次取出一个批次的消息批量处理。

rabbitmq：比较大

zeromq：极大

rocketmq：大

rocketMQ接收端可以批量消费消息，可以配置每次消费的消息数，但是发送端不是批量发送。

activemq：比较大

十二、订阅形式和消息分发

Kafka：基于topic以及按照topic进行正则匹配的发布订阅模式。

【发送】

发送端由topic和key来决定消息发往哪个分区，如果key为null，那么会使用轮询算法将消息均衡地发送到同一个topic的不同分区中。如果key不为null，那么会根据key的hashcode取模计算出要发往的分区。

【接收】

1>consumer向群组协调器broker发送心跳来维持他们和群组的从属关系以及他们对分区的所有权关系，所有权关系一旦被分配就不会改变除非发生再均衡(比如有一个consumer加入或者离开consumer group)，consumer只会从对应的分区读取消息。

2>kafka限制consumer个数要少于分区个数,每个消息只会被同一个 Consumer Group的一个 consumer消费（非广播）。

3>kafka的 Consumer Group订阅同一个topic，会尽可能地使得每一个consumer分配到相同数量的分区，不同 Consumer Group订阅同一个主题相互独立，同一个消息会被不同的 Consumer Group处理。

rabbitmq：提供了4种：direct, topic, Headers和fanout。

【发送】

先要声明一个队列，这个队列会被创建或者已经被创建，队列是基本存储单元。

由exchange和key决定消息存储在哪个队列。

direct>发送到和bindingKey完全匹配的队列。

topic>路由key是含有"."的字符串，会发送到含有"*"、"#"进行模糊匹配的bingKey对应的队列。

fanout>与key无关，会发送到所有和exchange绑定的队列

【接收】

rabbitmq的队列是基本存储单元，不再被分区或者分片，对于我们已经创建了的队列，消费端要指定从哪一个队列接收消息。

当rabbitmq队列拥有多个消费者的时候，队列收到的消息将以轮询的分发方式发送给消费者。每条消息只会发送给订阅列表里的一个消费者，不会重复。

这种方式非常适合扩展，而且是专门为并发程序设计的。

如果某些消费者的任务比较繁重，那么可以设置basicQos限制信道上消费者能保持的最大未确认消息的数量，在达到上限时，rabbitmq不再向这个消费者发送任何消息。

zeromq：点对点(p2p)

rocketmq：基于topic/messageTag以及按照消息类型、属性进行正则匹配的发布订阅模式

【发送】

发送消息通过轮询队列的方式发送，每个队列接收平均的消息量。发送消息指定topic、tags、keys，无法指定投递到哪个队列（没有意义，集群消费和广播消费跟消息存放在哪个队列没有关系）。

tags选填，类似于 Gmail 为每封邮件设置的标签，方便服务器过滤使用。目前只支持每个消息设置一个 tag，所以也可以类比为 Notify 的 MessageType 概念。

keys选填，代表这条消息的业务关键词，服务器会根据 keys 创建哈希索引，设置后，可以在 Console 系统根据 Topic、Keys 来查询消息，由于是哈希索引，请尽可能保证 key 唯一，例如订单号，商品 Id 等。

【接收】

1>广播消费。一条消息被多个Consumer消费，即使Consumer属于同一个ConsumerGroup，消息也会被ConsumerGroup中的每个Consumer都消费一次。

2>集群消费。一个 Consumer Group中的Consumer实例平均分摊消费消息。例如某个Topic有 9 条消息，其中一个Consumer Group有3个实例，那么每个实例只消费其中的 3 条消息。即每一个队列都把消息轮流分发给每个consumer。

activemq：点对点(p2p)、广播（发布-订阅）

点对点模式，每个消息只有1个消费者；

发布/订阅模式，每个消息可以有多个消费者。

【发送】

点对点模式：先要指定一个队列，这个队列会被创建或者已经被创建。

发布/订阅模式：先要指定一个topic，这个topic会被创建或者已经被创建。

【接收】

点对点模式：对于已经创建了的队列，消费端要指定从哪一个队列接收消息。



十三、顺序消息

Kafka：支持。

设置生产者的`max.in.flight.requests.per.connection`为1，可以保证消息是按照发送顺序写入服务器的，即使发生了重试。

kafka保证同一个分区里的消息是有序的，但是这种有序分两种情况

1>key为null，消息逐个被写入不同主机的分区中，但是对于每个分区依然是有序的

2>key不为null，消息被写入到同一个分区，这个分区的信息都是有序。

rabbitmq：不支持

zeromq：不支持

rocketmq：支持

activemq：不支持

十四、消息确认

Kafka：支持。

1>发送方确认机制

`ack=0`，不管消息是否成功写入分区

`ack=1`，消息成功写入首领分区后，返回成功

`ack=all`，消息成功写入所有分区后，返回成功。

2>接收方确认机制

自动或者手动提交分区偏移量，早期版本的kafka偏移量是提交给Zookeeper的，这样使得zookeeper的压力比较大，更新版本的kafka的偏移量是提交给kafka服务器的，不再依赖于zookeeper群组，集群的性能更加稳定。

rabbitmq：支持。

1>发送方确认机制，消息被投递到所有匹配的队列后，返回成功。如果消息和队列是可持久化的，那么在写入磁盘后，返回成功。支持批量确认和异步确认。

2>接收方确认机制，设置`autoAck`为false，需要显式确认，设置`autoAck`为true，自动确认。

当`autoAck`为false的时候，rabbitmq队列会分成两部分，一部分是等待投递给consumer的消息，一部分是已经投递但是没收到确认的消息。如果一直没有收到确认信号，并且consumer已经断开连接，rabbitmq会安排这个消息重新进入队列，投递给原来的消费者或者下一个消费者。

未确认的消息不会有过期时间，如果一直没有确认，并且没有断开连接，rabbitmq会一直等待，rabbitmq允许一条消息处理的时间可以很久很久。

zeromq：支持。

rocketmq：支持。

activemq：支持。

十五、消息回溯

Kafka：支持指定分区offset位置的回溯。

rabbitmq：不支持



rocketmq：支持指定时间点的回溯。

activemq：不支持

十六、消息重试

Kafka：不支持，但是可以实现。

kafka支持指定分区offset位置的回溯，可以实现消息重试。

rabbitmq：不支持，但是可以利用消息确认机制实现。

rabbitmq接收方确认机制，设置autoAck为false。

当autoAck为false的时候，rabbitmq队列会分成两部分，一部分是等待投递给consumer的消息，一部分是已经投递但是没收到确认的消息。如果一直没有收到确认信号，并且consumer已经断开连接，rabbitmq会安排这个消息重新进入队列，投递给原来的消费者或者下一个消费者。

zeromq：不支持，

rocketmq：支持。

消息消费失败的大部分场景下，立即重试99%都会失败，所以rocketmq的策略是在消费失败时定时重试，每次时间间隔相同。

1>发送端的 send 方法本身支持内部重试，重试逻辑如下：

a)至多重试3次；

b)如果发送失败，则轮转到下一个broker；

c)这个方法的总耗时不超过sendMsgTimeout 设置的值，默认 10s，超过时间不在重试。

2>接收端。

Consumer 消费消息失败后，要提供一种重试机制，令消息再消费一次。Consumer 消费消息失败通常可以分为以下两种情况：

\1. 由于消息本身的原因，例如反序列化失败，消息数据本身无法处理（例如话费充值，当前消息的手机号被

注销，无法充值）等。定时重试机制，比如过 10s 秒后再重试。

\2. 由于依赖的下游应用服务不可用，例如 db 连接不可用，外系统网络不可达等。

即使跳过当前失败的消息，消费其他消息同样也会报错。这种情况可以 sleep 30s，再消费下一条消息，减轻 Broker 重试消息的压力。

activemq：不支持

十七、并发度

Kafka：高

一个线程一个消费者，kafka限制消费者的个数要小于等于分区数，如果要提高并行度，可以在消费者中再开启多线程，或者增加consumer实例数量。

rabbitmq：极高

本身是用Erlang语言写的，并发性能高。

当rabbitmq队列拥有多个消费者的时候，队列收到的消息将以轮询的分发方式发送给消费者。每条消息只会发送给订阅列表里的一个消费者，不会重复。

这种方式非常适合扩展，而且是专门为并发程序设计的。

如果某些消费者的任务比较繁重，那么可以设置basicQos限制信道上消费者能保持的最大未确认消息的数量，在达到上限时，rabbitmq不再向这个消费者发送任何消息。

zeromq：高

rocketmq：高

1>rocketmq限制消费者的个数少于等于队列数，但是可以在消费者中再开启多线程，这一点和kafka是一致的，提高并行度的方法相同。

修改消费并行度方法

a) 同一个 ConsumerGroup 下，通过增加 Consumer 实例数量来提高并行度，超过订阅队列数的 Consumer实例无效。

b) 提高单个 Consumer 的消费并行线程，通过修改参数consumeThreadMin、consumeThreadMax

2>同一个网络连接connection，客户端多个线程可以同时发送请求，连接会被复用，减少性能开销。

activemq：高

附录二：Kafka调优

1、网络和io操作线程配置优化

broker处理消息的最大线程数（默认为3）

num.network.threads=cpu核数+1

broker处理磁盘IO的线程数

num.io.threads=cpu核数*2

2、log数据文件刷盘策略

每当producer写入10000条消息时，刷数据到磁盘

log.flush.interval.messages=10000

每间隔1秒钟时间，刷数据到磁盘

log.flush.interval.ms=1000

3、日志保留策略配置

保留三天，也可以更短（log.cleaner.delete.retention.ms）

log.retention.hours=72

段文件配置1GB，有利于快速回收磁盘空间，重启kafka加载也会加快(如果文件过小，则文件数量比较多，kafka启动时是单线程扫描目录(log.dir)下所有数据文件

log.segment.bytes=1073741824

这个参数指新创建一个topic时，默认的Replica数量,Replica过少会影响数据的可用性，太多则会白白浪费存储资源，一般建议在2~3为宜。

5、调优vm.max_map_count参数。

主要适用于Kafka broker上的主题数超多的情况。Kafka日志段的索引文件是用映射文件的机制来做的，故如果有超多日志段的话，这种索引文件数必然是很多的，极易打爆这个资源限制，所以对于这种情况一般要适当调大这个参数。

6、JVM堆大小

首先鉴于目前Kafka新版本已经不支持Java7了，而Java 8本身不更新了，甚至Java9其实都不做了，直接做Java10了，所以我建议Kafka至少搭配Java8来搭建。至于堆的大小，个人认为6-10G足矣。如果出现了堆溢出，就提jira给社区，让他们看到底是怎样的问题。因为这种情况下即使用户调大heap size，也只是延缓OOM而已，不太可能从根本上解决问题。