分布式消息服务 Kafka 版

最佳实践

文档版本 01

发布日期 2024-04-17

华为云计算技术有限公司

商标声明


和其他华为商标均为华为技术有限公司的商标。
本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受华为云计算技术有限公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，华为云计算技术有限公司对本文档内容不做任何明示或暗示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

目 录

1 DMS for Kafka 消费者 poll 的优化.....1
2 如何提高消息处理效率.....9
3 Kafka 业务迁移.....11
4 使用 MirrorMaker 跨集群数据同步.....14
5 Kafka 客户端参数配置建议.....18
6 Kafka 客户端使用规范.....21
7 如何设置消息堆积数超过阈值时，发送告警短信/邮件.....23
8 Logstash 对接 Kafka.....29
9 消息堆积最佳实践.....37
10 业务过载最佳实践.....39
11 业务数据不均衡最佳实践.....41
12 Python 调用 SDK 生产消息和查询消息.....43

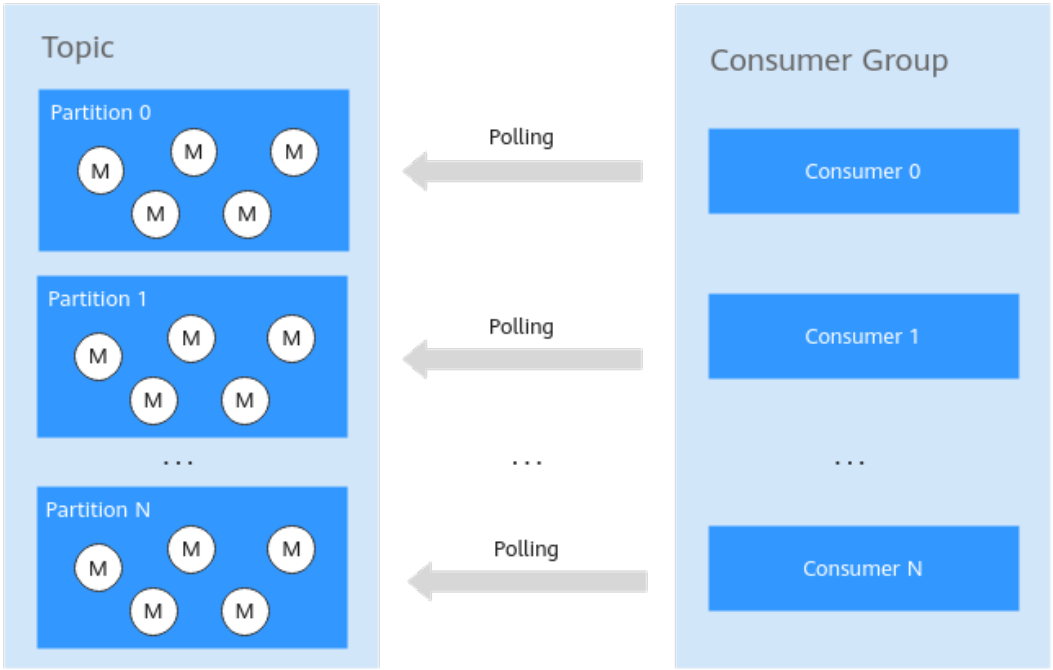
1 DMS for Kafka 消费者 poll 的优化

场景介绍

在DMS for Kafka提供的原生Kafka SDK中，消费者可以自定义拉取消息的时长，如果需要长时间的拉取消息，只需要把poll(long)方法的参数设置合适的值即可。但是这样的长连接可能会对客户端和服务端造成一定的压力，特别是分区数较多且每个消费者开启多个线程的情况下。

如图1-1所示，Topic含有多个分区，消费组中有多个消费者同时进行消费，每个线程均为长连接。当Topic中消息较少或者没有时，连接不断开，所有消费者不间断地拉取消息，这样造成了一定的资源浪费。

图 1-1 Kafka 消费者多线程消费模式



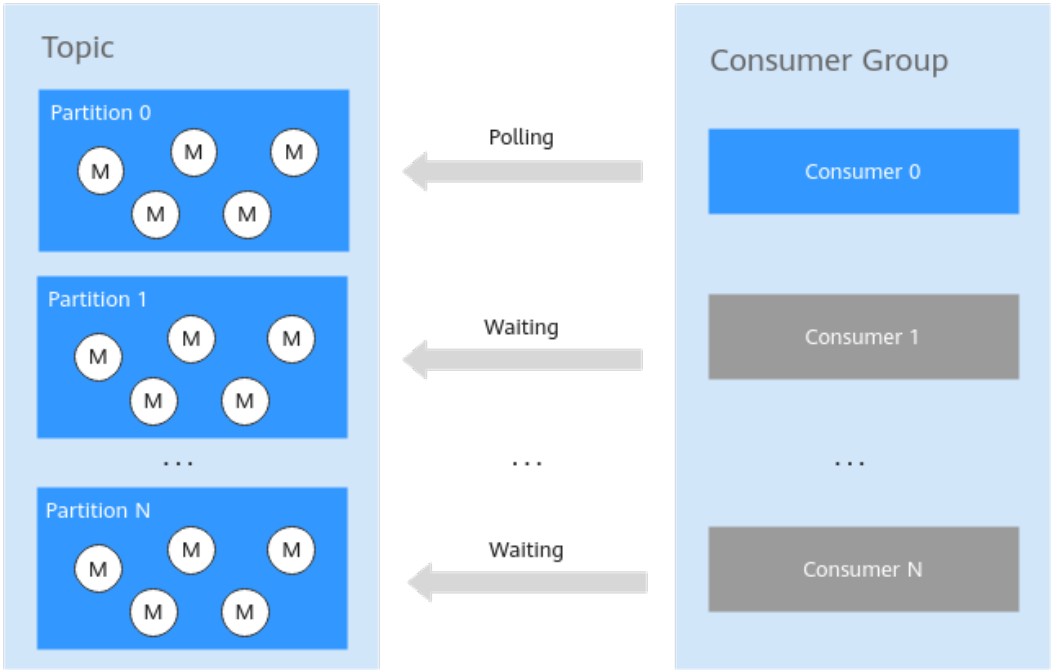
优化方案

在开了多个线程同时访问的情况下，如果Topic里已经没有消息了，其实不需要所有的线程都在poll，只需要有一个线程poll各分区的消息就足够了，当在polling的线程发现

Topic中有消息，可以唤醒其他线程一起消费消息，以达到快速响应的目的。如图1-2所示。

这种方案适用于对消费消息的实时性要求不高的应用场景。如果要求准实时消费消息，则建议保持所有消费者处于活跃状态。

图 1-2 优化后的多线程消费方案



说明

消费者（Consumer）和消息分区（Partition）并不强制数量相等，Kafka的poll(long)方法帮助实现获取消息、分区平衡、消费者与Kafka broker节点间的心跳检测等功能。

因此在对消费消息的实时性要求不高场景下，当消息数量不多的时候，可以选择让一部分消费者处于wait状态。

代码示例

须知

以下仅贴出与消费者线程唤醒与睡眠相关代码，如需运行整个demo，请先下载完整的**代码示例包**，同时参考**开发指南**进行部署和运行。

消费消息代码示例如下：

```
package com.huawei.dms.kafka;

import java.io.IOException;
import java.util.Arrays;
import java.util.Collection;
import java.util.Iterator;
import java.util.Properties;

import org.apache.kafka.clients.consumer.ConsumerRebalanceListener;
import org.apache.kafka.clients.consumer.ConsumerRecord;
```

```
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.TopicPartition;
import org.apache.log4j.Logger;

public class DmsKafkaConsumeDemo
{
    private static Logger logger = Logger.getLogger(DmsKafkaProduceDemo.class);

    public static void WorkerFunc(int workerId, KafkaConsumer<String, String> kafkaConsumer) throws
IOException
    {
        Properties consumerConfig = Config.getConsumerConfig();
        RecordReceiver receiver = new RecordReceiver(workerId, kafkaConsumer,
consumerConfig.getProperty("topic"));
        while (true)
        {
            ConsumerRecords<String, String> records = receiver.receiveMessage();
            Iterator<ConsumerRecord<String, String>> iter = records.iterator();
            while (iter.hasNext())
            {
                ConsumerRecord<String, String> cr = iter.next();
                System.out.println("Thread" + workerId + " recievedrecords" + cr.value());
                logger.info("Thread" + workerId + " recievedrecords" + cr.value());

            }

        }
    }

    public static KafkaConsumer<String, String> getConsumer() throws IOException
    {
        Properties consumerConfig = Config.getConsumerConfig();

        consumerConfig.put("ssl.truststore.location", Config.getTrustStorePath());
        System.setProperty("java.security.auth.login.config", Config.getSaslConfig());

        KafkaConsumer<String, String> kafkaConsumer = new KafkaConsumer<>(consumerConfig);
        kafkaConsumer.subscribe(Arrays.asList(consumerConfig.getProperty("topic")),
                new ConsumerRebalanceListener()
                {
                    @Override
                    public void onPartitionsRevoked(Collection<TopicPartition> arg0)
                    {

                    }

                    @Override
                    public void onPartitionsAssigned(Collection<TopicPartition> tps)
                    {

                    }
                });
        return kafkaConsumer;
    }

    public static void main(String[] args) throws IOException
    {

        //创建当前消费组的consumer
        final KafkaConsumer<String, String> consumer1 = getConsumer();
        Thread thread1 = new Thread(new Runnable()
        {
            public void run()
            {
                try
                {
                    WorkerFunc(1, consumer1);
                }
```



```
            catch (IOException e)
            {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    });
    final KafkaConsumer<String, String> consumer2 = getConsumer();

    Thread thread2 = new Thread(new Runnable()
    {
        public void run()
        {
            try
            {
                WorkerFunc(2, consumer2);
            }
            catch (IOException e)
            {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    });
    final KafkaConsumer<String, String> consumer3 = getConsumer();

    Thread thread3 = new Thread(new Runnable()
    {
        public void run()
        {
            try
            {
                WorkerFunc(3, consumer3);
            }
            catch (IOException e)
            {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    });

    //启动线程
    thread1.start();
    thread2.start();
    thread3.start();

    try
    {
        Thread.sleep(5000);
    }
    catch (InterruptedException e)
    {
        e.printStackTrace();
    }
    //线程加入
    try
    {
        thread1.join();
        thread2.join();
        thread3.join();
    }
    catch (InterruptedException e)
    {
        e.printStackTrace();
    }
  }
}
```

消费者线程管理示例代码

示例仅提供简单的设计思路，开发者可结合实际场景优化线程休眠和唤醒机制。

```
package com.huawei.dms.kafka;

import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;

import org.apache.log4j.Logger;

public class RecordReceiver
{
    private static Logger logger = Logger.getLogger(DmsKafkaProduceDemo.class);

    //polling的间隔时间
    public static final int WAIT_SECONDS = 10 * 1000;

    protected static final Map<String, Object> sLockObjMap = new HashMap<String, Object>();

    protected static Map<String, Boolean> sPollingMap = new ConcurrentHashMap<String, Boolean>();

    protected Object lockObj;

    protected String topicName;

    protected KafkaConsumer<String, String> kafkaConsumer;

    protected int workerId;

    public RecordReceiver(int id, KafkaConsumer<String, String> kafkaConsumer, String queue)
    {
        this.kafkaConsumer = kafkaConsumer;
        this.topicName = queue;
        this.workerId = id;

        synchronized (sLockObjMap)
        {
            lockObj = sLockObjMap.get(topicName);
            if (lockObj == null)
            {
                lockObj = new Object();
                sLockObjMap.put(topicName, lockObj);
            }
        }
    }

    public boolean setPolling()
    {
        synchronized (lockObj)
        {
            Boolean ret = sPollingMap.get(topicName);
            if (ret == null || !ret)
            {
                sPollingMap.put(topicName, true);
                return true;
            }
            return false;
        }
    }

    //唤醒全部线程
    public void clearPolling()
    {
        synchronized (lockObj)
        {
            sPollingMap.put(topicName, false);
            lockObj.notifyAll();
```

```
            System.out.println("Everyone WakeUp and Work!");
            logger.info("Everyone WakeUp and Work!");
        }
    }

    public ConsumerRecords<String, String> receiveMessage()
    {
        boolean polling = false;
        while (true)
        {
            //检查线程的poll状态，必要时休眠
            synchronized (lockObj)
            {
                Boolean p = sPollingMap.get(topicName);
                if (p != null && p)
                {
                    try
                    {
                        System.out.println("Thread" + workerId + " Have a nice sleep!");
                        logger.info("Thread" + workerId +" Have a nice sleep!");
                        polling = false;
                        lockObj.wait();
                    }
                    catch (InterruptedException e)
                    {
                        System.out.println("MessageReceiver Interrupted! topicName is " + topicName);
                        logger.error("MessageReceiver Interrupted! topicName is "+topicName);

                        return null;
                    }
                }
            }

            //开始消费，必要时唤醒其他线程消费
            try
            {
                ConsumerRecords<String, String> Records = null;
                if (!polling)
                {
                    Records = kafkaConsumer.poll(100);
                    if (Records.count() == 0)
                    {
                        polling = true;
                        continue;
                    }
                }
                else
                {
                    if (setPolling())
                    {
                        System.out.println("Thread" + workerId + " Polling!");
                        logger.info("Thread " + workerId + " Polling!");
                    }
                    else
                    {
                        continue;
                    }
                    do
                    {
                        System.out.println("Thread" + workerId + " KEEP Poll records!");
                        logger.info("Thread" + workerId + " KEEP Poll records!");
                        try
                        {
                            Records = kafkaConsumer.poll(WAIT_SECONDS);
                        }
                        catch (Exception e)
                        {
                            System.out.println("Exception Happened when polling records: " + e);
                            logger.error("Exception Happened when polling records: " + e);
```



```
                }
            } while (Records.count()==0);
            clearPolling();
        }
        //消息确认
        kafkaConsumer.commitSync();
        return Records;
    }
    catch (Exception e)
    {
        System.out.println("Exception Happened when poll records: " + e);
        logger.error("Exception Happened when poll records: " + e);
    }
  }
 }
}
```

说明

topicName配置为Topic名称。

示例代码运行结果

```
[2018-01-25 22:40:51,841] INFO Thread 2 Polling! (com.huawei.dms.kafka.DmsKafkaProduceDemo:119)
[2018-01-25 22:40:51,841] INFO Thread2 KEEP Poll records!
(com.huawei.dms.kafka.DmsKafkaProduceDemo:128)
[2018-01-25 22:40:52,122] INFO Everyone WakeUp and Work!
(com.huawei.dms.kafka.DmsKafkaProduceDemo:69)
[2018-01-25 22:40:52,169] INFO Thread2 recievedrecordshello, dms kafka.
(com.huawei.dms.kafka.DmsKafkaProduceDemo:32)
[2018-01-25 22:40:52,169] INFO Thread2 recievedrecordshello, dms kafka.
(com.huawei.dms.kafka.DmsKafkaProduceDemo:32)
[2018-01-25 22:40:52,216] INFO Thread2 recievedrecordshello, dms kafka.
(com.huawei.dms.kafka.DmsKafkaProduceDemo:32)
[2018-01-25 22:40:52,325] INFO Thread 2 Polling! (com.huawei.dms.kafka.DmsKafkaProduceDemo:119)
[2018-01-25 22:40:52,325] INFO Thread2 KEEP Poll records!
(com.huawei.dms.kafka.DmsKafkaProduceDemo:128)
[2018-01-25 22:40:54,947] INFO Thread1 Have a nice sleep!
(com.huawei.dms.kafka.DmsKafkaProduceDemo:87)
[2018-01-25 22:40:54,979] INFO Thread3 Have a nice sleep!
(com.huawei.dms.kafka.DmsKafkaProduceDemo:87)
[2018-01-25 22:41:32,347] INFO Thread2 KEEP Poll records!
(com.huawei.dms.kafka.DmsKafkaProduceDemo:128)
[2018-01-25 22:41:42,353] INFO Thread2 KEEP Poll records!
(com.huawei.dms.kafka.DmsKafkaProduceDemo:128)
[2018-01-25 22:41:47,816] INFO Everyone WakeUp and Work!
(com.huawei.dms.kafka.DmsKafkaProduceDemo:69)
[2018-01-25 22:41:47,847] INFO Thread2 recievedrecordshello, dms kafka.
(com.huawei.dms.kafka.DmsKafkaProduceDemo:32)
[2018-01-25 22:41:47,925] INFO Thread 3 Polling! (com.huawei.dms.kafka.DmsKafkaProduceDemo:119)
[2018-01-25 22:41:47,925] INFO Thread1 Have a nice sleep!
(com.huawei.dms.kafka.DmsKafkaProduceDemo:87)
[2018-01-25 22:41:47,925] INFO Thread3 KEEP Poll records!
(com.huawei.dms.kafka.DmsKafkaProduceDemo:128)
[2018-01-25 22:41:47,957] INFO Thread2 Have a nice sleep!
(com.huawei.dms.kafka.DmsKafkaProduceDemo:87)
[2018-01-25 22:41:48,472] INFO Everyone WakeUp and Work!
(com.huawei.dms.kafka.DmsKafkaProduceDemo:69)
[2018-01-25 22:41:48,503] INFO Thread3 recievedrecordshello, dms kafka.
(com.huawei.dms.kafka.DmsKafkaProduceDemo:32)
[2018-01-25 22:41:48,518] INFO Thread1 recievedrecordshello, dms kafka.
(com.huawei.dms.kafka.DmsKafkaProduceDemo:32)
[2018-01-25 22:41:48,550] INFO Thread2 recievedrecordshello, dms kafka.
(com.huawei.dms.kafka.DmsKafkaProduceDemo:32)
[2018-01-25 22:41:48,597] INFO Thread1 recievedrecordshello, dms kafka.
(com.huawei.dms.kafka.DmsKafkaProduceDemo:32)
[2018-01-25 22:41:48,659] INFO Thread 2 Polling! (com.huawei.dms.kafka.DmsKafkaProduceDemo:119)
```



```
[2018-01-25 22:41:48,659] INFO Thread2 KEEP Poll records!
(com.huawei.dms.kafka.DmsKafkaProduceDemo:128)
[2018-01-25 22:41:48,675] INFO Thread3 recievedrecordshello, dms kafka.
(com.huawei.dms.kafka.DmsKafkaProduceDemo:32)
[2018-01-25 22:41:48,675] INFO Everyone WakeUp and Work!
(com.huawei.dms.kafka.DmsKafkaProduceDemo:69)
[2018-01-25 22:41:48,706] INFO Thread 1 Polling! (com.huawei.dms.kafka.DmsKafkaProduceDemo:119)
[2018-01-25 22:41:48,706] INFO Thread1 KEEP Poll records!
(com.huawei.dms.kafka.DmsKafkaProduceDemo:128)
```

2 如何提高消息处理效率

消息发送和消费的可靠性必须由分布式消息服务Kafka版和生产者以及消费者协同工作才能保证。同时开发者需要尽量合理使用分布式消息服务Kafka版的Topic，以提高消息发送和消息消费的效率与准确性。

对使用分布式消息服务Kafka版的生产者和消费者有如下的使用建议：

重视消息生产与消费的确认过程

消息生产（发送）

发送消息后，生产者需要根据分布式消息服务Kafka版的返回信息确认消息是否发送成功，如果返回失败需要重新发送。

生产消息时，生产者通过同步等待发送结果或异步回调函数，判断消息是否发送成功。在消息传递过程中，如果发生异常，生产者没有接收到发送成功的信号，生产者自己决策是否需要重复发送消息。如果接收到发送成功的信号，则表明该消息已经被分布式消息服务Kafka版可靠存储。

消息消费

消息消费时，消费者需要确认消息是否已被成功消费。

生产的消息被依次存储在分布式消息服务Kafka版的存储介质中。消费时依次获取分布式消息服务Kafka版中存储的消息。消费者获取消息后，进行消费并记录消费成功或失败的状态，并将消费状态提交到分布式消息服务Kafka版。

在消费过程中，如果出现异常，没有提交消费确认，该批消息会在后续的消费请求中再次被获取。

消息生产与消费的幂等传递

分布式消息服务Kafka版设计了一系列可靠性保障措施，确保消息不丢失。例如使用消息同步存储机制防止系统与服务器层面的异常重启或者掉电，使用消息确认（ACK）机制解决消息传输过程中遇到的异常。

考虑到网络异常等极端情况，用户除了做好消息生产与消费的确认，还需要配合分布式消息服务Kafka版完成消息发送与消费的重复传输设计。

- 当无法确认消息是否已发送成功，生产者需要将消息重复发送给分布式消息服务Kafka版。

- 当重复收到已处理过的消息，消费者需要告诉分布式消息服务Kafka版消费成功且保证不重复处理。

消息可以批量生产和消费

为提高消息发送和消息消费效率，推荐使用批量消息发送和消费。

图 2-1 消息批量生产（发送）与消费

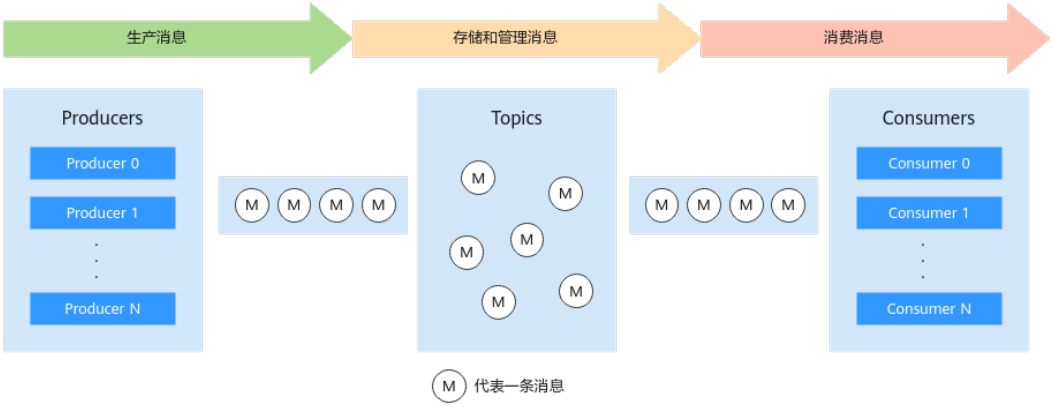
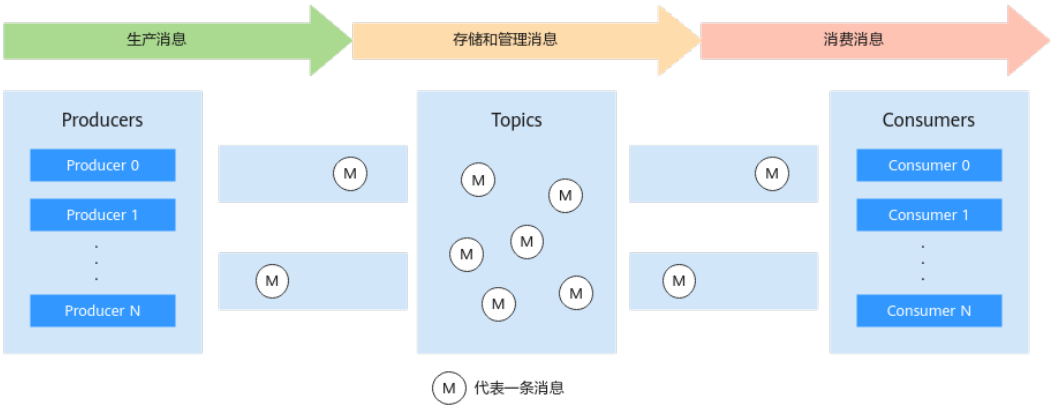


图 2-2 消息逐条生产（发送）与消费



此外，批量消费消息时，消费者应按照接收的顺序对消息进行处理、确认，当对某一条消息处理失败时，不再需要继续处理本批消息中的后续消息，直接对已正确处理的消息进行确认即可。

巧用消费组协助运维

用户使用分布式消息服务Kafka版作为消息中间件，查看Topic的消息内容对于定位问题与调试服务是至关重要的。

当消息的生产和消费过程中遇到疑难问题时，通过创建不同消费组可以帮助定位分析问题或调试服务对接。用户可以创建一个新的消费组，对Topic中的消息进行消费并分析消费过程，这样不会影响其他服务对消息的处理。

3 Kafka 业务迁移

应用场景

Kafka迁移指将生产与消费消息的客户端切换成连接新Kafka，部分还涉及将持久化的消息文件迁移到新的Kafka。主要涉及到以下2类场景：

- 业务上云且不希望业务有中断。

  在上云过程中，连续性要求高的业务，需要平滑迁移，不能有长时间的中断。
- 在云上变更业务部署

  单AZ部署的Kafka实例，不具备AZ之间的容灾能力。用户对可靠性要求提升后，需要迁移到多AZ部署的实例上。

约束与限制

- 使用Smart Connect迁移业务，会对源端Kafka进行消费，对目标端Kafka进行生产，会占用源端和目标端Kafka的带宽。
- 出于性能考虑，Smart Connect实时同步源端和目标端的数据，但是消费进度是通过批处理同步的，可能会导致源端和目标端每个分区的消费进度存在0-100之间的差异。

迁移准备

1. 配置网络环境

   Kafka实例分内网地址以及公网地址两种网络连接方式。如果使用公网地址，则消息生成与消费客户端需要有公网访问权限，并配置如下安全组。

   表 3-1 安全组规则

方向	协议	端口	源地址	说明
入方向	TCP	9094	0.0.0.0/0	通过公网访问Kafka（关闭SSL加密）。
入方向	TCP	9095	0.0.0.0/0	通过公网访问Kafka（开启SSL加密）。

2. 创建Kafka实例

Kafka的规格不能低于原业务使用的Kafka规格。具体请参考**创建Kafka实例**。

3. 创建Topic

   在新的Kafka实例上创建与原Kafka实例相同配置的Topic，包括Topic名称、副本数、分区数、消息老化时间，以及是否同步复制和落盘等。具体请参考**创建Topic**。

实施步骤（方案一：先迁生产，再迁消费）

指先将生产消息的业务迁移到新的Kafka，原Kafka不会有新的消息生产。待原有Kafka实例的消息全部消费完成后，再将消费消息业务迁移到新的Kafka，开始消费新Kafka实例的消息。

步骤1 将生产客户端的Kafka连接地址修改为新Kafka实例的连接地址。

步骤2 重启生产业务，使得生产者将新的消息发送到新Kafka实例中。

步骤3 观察各消费组在原Kafka的消费进度，直到原Kafka中数据都已经被消费完毕。

步骤4 将消费客户端的Kafka连接地址修改为新Kafka实例的连接地址。

步骤5 重启消费业务，使得消费者从新Kafka实例中消费消息。

步骤6 观察消费者是否能正常从新Kafka实例中获取数据。

步骤7 迁移结束。

----结束

本方案为业界通用的迁移方案，操作步骤简单，迁移过程由业务侧自主控制，整个过程中消息不会存在乱序问题，**适用于对消息顺序有要求的场景**。但是该方案中需要等待消费者业务直至消费完毕，存在一个时间差的问题，部分数据可能存在较大的端到端时延。

实施步骤（方案二：同时消费，后迁生产）

指消费者业务启用多个消费客户端，分别向原Kafka和新Kafka实例消费消息，然后将生产业务切到新Kafka实例，这样能确保所有消息都被及时消费。

步骤1 启动新的消费客户端，配置Kafka连接地址为新Kafka实例的连接地址，消费新Kafka实例中的数据。

说明

原有消费客户端需继续运行，消费业务同时消费原Kafka与新Kafka实例的消息。

步骤2 修改生产客户端，Kafka连接地址改为新Kafka实例的连接地址。

步骤3 重启生产客户端，将生产业务迁移到新Kafka实例中。

步骤4 生产业务迁移后，观察连接新Kafka实例的消费业务是否正常。

步骤5 等待原Kafka中数据消费完毕，关闭原有消费业务客户端。

步骤6 迁移结束。

----结束

迁移过程由业务自主控制。本方案中消费业务会在一段时间内同时消费原Kafka和新Kafka实例。由于在迁移生产业务之前，已经有消费业务运行在新Kafka实例上，因此

不会存在端到端时延的问题。但在迁移生产的开始阶段，同时消费原Kafka与新Kafka实例，会导致部分消息之间的生产顺序无法保证，存在消息乱序的问题。此场景**适用于对端到端时延有要求，却对消息顺序不敏感的业务**。

实施步骤（方案三：先迁消费，再迁生产）

指首先通过Smart Connect同步两个Kafka的消息，其次将消费端迁移到新Kafka，最后将生产端迁移到新Kafka。

步骤1 创建Kafka数据复制的Smart Connect任务，用于同步两个Kafka的消息。具体步骤请参见**创建Smart Connect任务（Kafka数据复制）**。

步骤2 在Kafka控制台的“消息查询”页面，查看两个Kafka的最新消息是否一致，确认两个Kafka的同步进度是否一致。具体步骤请参见**查询消息**。

- 是，执行**步骤3**。
- 否，在监控页面查看两个Kafka的“Kafka每分钟同步数据量”是否正常，如果正常，先等待两个Kafka的同步进度一致，然后执行**步骤3**。

步骤3 将消费客户端的Kafka连接地址修改为新Kafka实例的连接地址。

步骤4 重启消费业务，使得消费者从新Kafka实例中消费消息。

步骤5 观察消费者是否能正常从新Kafka实例中获取数据。

步骤6 修改生产客户端，Kafka连接地址改为新Kafka实例的连接地址。

步骤7 重启生产客户端，将生产业务迁移到新Kafka实例中。

步骤8 生产业务迁移后，观察连接新Kafka实例的消费业务是否正常。

步骤9 迁移结束。

----结束

本方案依赖于Smart Connect，Smart Connect实时同步源端和目标端的数据，但是消费进度是通过批处理同步的，可能会导致源端和目标端每个分区的消费进度存在0-100之间的差异，存在少量重复消费问题。此场景适用于生产端不可停止，端到端有时延要求，但是可以兼容少量重复消费的业务。

常见问题：如何将持久化数据也一起迁移

如果需要将原Kafka的已消费数据也迁移到Kafka实例，可以使用Smart Connect工具，模拟成原Kafka的消费客户端，以及新Kafka实例的生产客户端，将Kafka所有消息数据迁移到新的Kafka实例，具体步骤请参考**创建Smart Connect任务（Kafka数据复制）**。

需要注意的是，华为云Kafka实例为3副本存储，因此建议实例存储空间为原业务的单副本消息存储的3倍。

4 使用 MirrorMaker 跨集群数据同步

应用场景

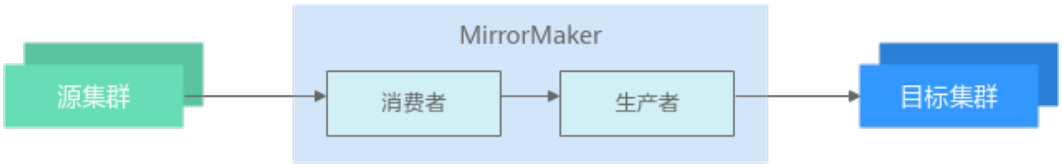
在以下场景，使用MirrorMaker进行不同集群间的数据同步，可以确保Kafka集群的可用性和可靠性。

- 备份和容灾：企业存在多个数据中心，为了防止其中一个数据中心出现问题，导致业务不可用，会将集群数据同步备份在多个不同的数据中心。
- 集群迁移：当今很多企业将业务迁移上云，迁移过程中需要确保线下集群和云上集群的数据同步，保证业务的连续性。

方案架构

使用MirrorMaker可以实现将源集群中的数据镜像复制到目标集群中。其原理如图4-1所示，MirrorMaker本质上也是生产消费消息，首先从源集群中消费数据，然后将消费的数据生产到目标集群。如果您需要了解更多关于MirrorMaker的信息，请参见**Mirroring data between clusters**。

图 4-1 MirrorMaker 原理图



约束与限制

- 源集群中节点的IP地址和端口号不能和目标集群中节点的IP地址和端口号相同，否则会导致数据在Topic内无限循环复制。
- 使用MirrorMaker同步数据，至少需要有两个或以上集群，不可在单个集群内部使用MirrorMaker，否则会导致数据在Topic内无限循环复制。

实施步骤

步骤1 购买一台弹性云服务器，确保弹性云服务器与源集群、目标集群网络互通。具体购买操作，请参考**购买弹性云服务器**。

步骤2 登录弹性云服务器，安装Java JDK，并配置JAVA_HOME与PATH环境变量，使用执行用户在用户家目录下修改“.bash_profile”，添加如下行。其中“/opt/java/jdk1.8.0_151”为JDK的安装路径，请根据实际情况修改。

```
export JAVA_HOME=/opt/java/jdk1.8.0_151
export PATH=$JAVA_HOME/bin:$PATH
```

执行**source .bash_profile**命令使修改生效。

说明

弹性云服务器默认自带的JDK可能不符合要求，例如OpenJDK，需要配置为Oracle的JDK，可至Oracle官方下载页面**下载Java Development Kit 1.8.111及以上版本**。

步骤3 下载Kafka 3.3.1版本的二进制软件包。

```
wget https://archive.apache.org/dist/kafka/3.3.1/kafka_2.12-3.3.1.tgz
```

步骤4 解压二进制软件包。

```
tar -zxvf kafka_2.12-3.3.1.tgz
```

步骤5 进入二进制软件包目录，修改“config”目录下的“connect-mirror-maker.properties”的配置文件，在配置文件中指定源集群和目标集群的IP地址和端口以及其他配置。

```
# 指定两个集群
clusters = A, B
A.bootstrap.servers = A_host1:A_port, A_host2:A_port, A_host3:A_port
B.bootstrap.servers = B_host1:B_port, B_host2:B_port, B_host3:B_port

# 指定数据同步方向，可以单向同步也可互相同步
A->B.enabled = true

# 指定同步的Topic，支持正则匹配，默认复制全部Topic，如："foo-.*"
A->B.topics = .*

# 打开以下两个配置则表示A、B两个集群互相复制同步
#B->A.enabled = true
#B->A.topics = .*

# 设置副本个数，如果是要同步多个Topic且副本数各不相同，建议先创建同名同副本数的Topic再启动MirrorMaker
replication.factor=3

# 设置消费进度同步方向，可以单向同步也可互相同步
A->B.sync.group.offsets.enabled=true

############################# Internal Topic Settings  #############################
# The replication factor for mm2 internal topics "heartbeats", "B.checkpoints.internal" and
# "mm2-offset-syncs.B.internal"
# 测试环境可以为1，生产环境建议以下配置大于1，比如设为3
checkpoints.topic.replication.factor=3
heartbeats.topic.replication.factor=3
offset-syncs.topic.replication.factor=3

# The replication factor for connect internal topics "mm2-configs.B.internal", "mm2-offsets.B.internal" and
# "mm2-status.B.internal"
# 测试环境可以为1，生产环境建议以下配置大于1，比如设为3
offset.storage.replication.factor=3
status.storage.replication.factor=3
config.storage.replication.factor=3

# customize as needed
# replication.policy.separator = _
# sync.topic.acls.enabled = false
# emit.heartbeats.interval.seconds = 5
```

步骤6 在二进制软件包目录下，启动MirrorMaker，进行数据同步。

```
./bin/connect-mirror-maker.sh config/connect-mirror-maker.properties
```

步骤7 （可选）MirrorMaker开启后，如果在源集群上新建了Topic，如需对此Topic进行数据同步，则需重启MirrorMaker，重启步骤参考**步骤6**。也可配置自动同步新增Topic，按需增加如**表4-1**所示配置后，无需重启MirrorMaker，即可周期性同步新增Topic。其中，“refresh.topics.interval.seconds”为必选，其他参数根据实际情况选择。

表 4-1 MirrorMaker 配置参数

参数名	默认值	说明
sync.topic.configs.enabled	true	是否监控源集群的配置更改
sync.topic.acls.enabled	true	是否监控源集群ACL的更改
emit.heartbeats.enabled	true	连接器应定期发出心跳
emit.heartbeats.interval.seconds	5秒	心跳频率
emit.checkpoints.enabled	true	连接器应定期发出消费端偏移量信息
emit.checkpoints.interval.seconds	5秒	检查点的频率
refresh.topics.enabled	true	连接器应定期检查新主题
refresh.topics.interval.seconds	5秒	检查源群集中是否有新主题的频率
refresh.groups.enabled	true	连接器应定期检查新的消费组
refresh.groups.interval.seconds	5秒	检查源集群新的消费组频率
replication.policy.class	org.apache.kafka.connect.mirror.DefaultReplicationPolicy	使用LegacyReplicationPolicy模仿旧版MirrorMaker
heartbeats.topic.retention.ms	1天	首次创建心跳主题时使用
checkpoints.topic.retention.ms	1天	首次创建检查点主题时使用
offset.syncs.topic.retention.ms	max long	首次创建偏移同步主题时使用

----结束

验证数据是否同步

步骤1 在目标集群中查看Topic列表，确认是否有源集群Topic。

说明

目标集群中的Topic名称和源集群相比，多了前缀（如A.），这属于正常情况，是MirrorMaker 2为了防止Topic循环备份进行的设置。

步骤2 在源集群生产并消费消息，在目标集群查看消费进度，确认数据是否已从源集群同步到了目标集群。

如果目标集群为华为云Kafka实例的话，在分布式消息服务Kafka版控制台的“消费组管理 > 消费进度”中，查看消费进度。

----结束

5 Kafka 客户端参数配置建议

Kafka客户端的配置参数很多，以下提供Producer和Consumer几个常用参数配置。其他参数配置，请参考**Kafka配置**。

表 5-1 Producer 参数

参数	默认值	推荐值	说明
acks	1	高可靠：all或者-1 高吞吐：1	收到Server端确认信号个数，表示procuder需要收到多少个这样的确认信号，算消息发送成功。acks参数代表了数据备份的可用性。常用选项： acks=**0**：表示producer不需要等待任何确认收到的信息，副本将立即加到socket buffer并认为已经发送。没有任何保障可以保证此种情况下server已经成功接收数据，同时重试配置不会发生作用（因为客户端不知道是否失败）回馈的offset会总是设置为-1。 acks=**1**：这意味着至少要等待leader已经成功将数据写入本地log，但是并没有等待所有follower是否成功写入。如果follower没有成功备份数据，而此时leader又无法提供服务，则消息会丢失。 acks=**all**或者**-1**：这意味着leader需要等待ISR中所有备份都成功写入日志。只要任何一个备份存活，数据都不会丢失。min.insync.replicas指定必须确认写入才能被认为成功的副本的最小数量。
retries	0	结合实际业务调整	客户端发送消息的重试次数。值大于0时，这些数据发送失败后，客户端会重新发送。 注意，这些重试与客户端接收到发送错误时的重试没有什么不同。允许重试将潜在的改变数据的顺序，如果这两个消息记录都是发送到同一个partition，则第一个消息失败第二个发送成功，则第二条消息会比第一条消息出现要早。 针对网络闪断场景，生产者建议配置重试能力，推荐重试次数retries=3，重试间隔retry.backoff.ms=1000。

参数	默认值	推荐值	说明
request.timeout.ms	30000	结合实际业务调整	设置一个请求最大等待时间（单位为ms），超过这个时间则会抛Timeout异常。 超时时间如果设置大一些，如127000（127秒），高并发的场景中，能减少发送失败的情况。
block.on.buffer.full	TRUE	TRUE	TRUE表示当我们内存用尽时，停止接收新消息记录或者抛出错误。 默认情况下，这个设置为TRUE。然而某些阻塞可能不值得期待，因此立即抛出错误更好。如果设置为false，则producer抛出一个异常错误：BufferExhaustedException
batch.size	16384	262144	默认的批量处理消息字节数上限。producer将试图批处理消息记录，以减少请求次数。这将改善client与server之间的性能。不会试图处理大于这个字节数的消息字节数。 发送到brokers的请求将包含多个批量处理，其中会包含对每个partition的一个请求。 较小的批量处理数值比较少用，并且可能降低吞吐量（0则会仅用批量处理）。较大的批量处理数值将会浪费更多内存空间，这样就需要分配特定批量处理数值的内存大小。
buffer.memory	33554432	67108864	producer可以用来缓存数据的内存大小。如果数据产生速度大于向broker发送的速度，producer会阻塞或者抛出异常，以“block.on.buffer.full”来表明。 这项设置将和producer能够使用的总内存相关，但并不是一个硬性的限制，因为不是producer使用的所有内存都是用于缓存。一些额外的内存会用于压缩（如果引入压缩机制），同样还有一些用于维护请求。

表 5-2 Consumer 参数

参数	默认值	推荐值	说明
auto.commit.enable	TRUE	FALSE	如果为真，consumer所fetch的消息的offset将会自动的同步到zookeeper。这项提交的offset将在进程无法提供服务时，由新的consumer使用。 约束： 设置为false后，需要先成功消费再提交，这样可以避免消息丢失。

参数	默认值	推荐值	说明
auto.offset.reset	latest	earliest	没有初始化offset或者offset被删除时，可以设置以下值： ● earliest：自动复位offset为最早 ● latest：自动复位offset为最新 ● none：如果没有发现offset，则向消费者抛出异常 ● anything else：向消费者抛出异常。 **说明** 如果将此配置设置为latest，新增分区时，生产者可能会在消费者重置初始偏移量之前开始向新增加的分区发送消息，从而导致部分消息丢失。
connections.max.idle.ms	600000	30000	空连接的超时时间（单位为ms），设置为30000可以在网络异常场景下减少请求卡顿的时间。

6 Kafka 客户端使用规范

consumer 使用规范

1. consumer的owner线程需确保不会异常退出，避免客户端无法发起消费请求，阻塞消费。
2. 确保处理完消息后再做消息commit，避免业务消息处理失败，无法重新拉取处理失败的消息。
3. 通常不建议对每条消息都进行commit，如果对每条消息都进行了commit，会导致OFFSET_COMMIT请求过多，进而导致CPU使用率过高。例如：如果一个消费请求拉取1000条消息，每条都commit，则commit请求TPS是消费的1000倍，消息体越小，这个比例越大。建议隔一定条数或时间，批量commit，或打开enable.auto.commit，这样设置会存在一个缺点，即在客户端故障时，可能丢失一部分缓存的消费进度，导致重复消费。请根据业务实际情况，设置批量commit。
4. consumer不能频繁加入和退出group，频繁加入和退出，会导致consumer频繁做rebalance，阻塞消费。
5. consumer数量不能超过topic分区数，否则会有consumer拉取不到消息。
6. consumer需周期poll，维持和server的心跳，避免心跳超时，导致consumer频繁加入和退出，阻塞消费。
7. consumer拉取的消息本地缓存应有大小限制，避免OOM（Out of Memory）。
8. consumer session设置为30秒，session.timeout.ms=30000。
9. Kafka不能保证消费重复的消息，业务侧需保证消息处理的幂等性。
10. 消费线程退出要调用consumer的close方法，避免同一个组的其他消费者阻塞sesstion.timeout.ms的时间。
11. 消费组名称开头不使用特殊字符（如#），使用特殊字符可能会导致云监控无法展示此消费组的监控数据。

producer 使用规范

1. 同步复制客户端需要配合使用：acks=all
2. 配置发送失败重试：retries=3
3. 发送优化：对于时延敏感的信息，设置linger.ms=0。对于时延不敏感的信息，设置linger.ms在100~1000之间。
4. 生产端的JVM内存要足够，避免内存不足导致发送阻塞。

5. 时间戳设置为与当地时间一致，避免时间戳为未来时间导致消息无法老化。

topic 使用规范

配置要求：推荐3副本，同步复制，最小同步副本数为2，且同步副本数不能等于topic副本数，否则宕机1个副本会导致无法生产消息。

创建方式：支持选择是否开启kafka自动创建Topic的开关。选择开启后，表示生产或消费一个未创建的Topic时，会自动创建一个包含3个分区和3个副本的Topic。

单topic最大分区数建议为100。

topic副本数为3。

其他建议

连接数限制：3000

消息大小：不能超过10MB

使用sasl_ssl协议访问Kafka：确保DNS具有反向解析能力，或者在hosts文件配置kafka所有节点ip和主机名映射，避免Kafka client做反向解析，阻塞连接建立。

磁盘容量申请超过业务量 * 副本数的2倍，即保留磁盘空闲50%左右。

业务进程JVM内存使用确保无频繁FGC，否则会阻塞消息的生产和消费。

7 如何设置消息堆积数超过阈值时，发送告警短信/邮件

操作场景

如果您想要在消费组的消息堆积数超过阈值时，通过短信/邮件及时收到通知信息，可以参考本章节设置告警通知。

您还可以参考本章节为分布式消息服务Kafka版的**其他监控指标**设置告警通知。

前提条件

已**购买Kafka实例**、**创建Topic**，并且已成功消费消息。

操作步骤

步骤1 登录分布式消息服务Kafka版控制台，单击待创建告警通知的实例名称，进入实例详情页。

步骤2 在左侧导航栏，选择“监控”，进入监控页面。

步骤3 在“消费组”页签，设置需要创建告警通知的消费组。

图 7-1 选择需要创建告警通知的消费组

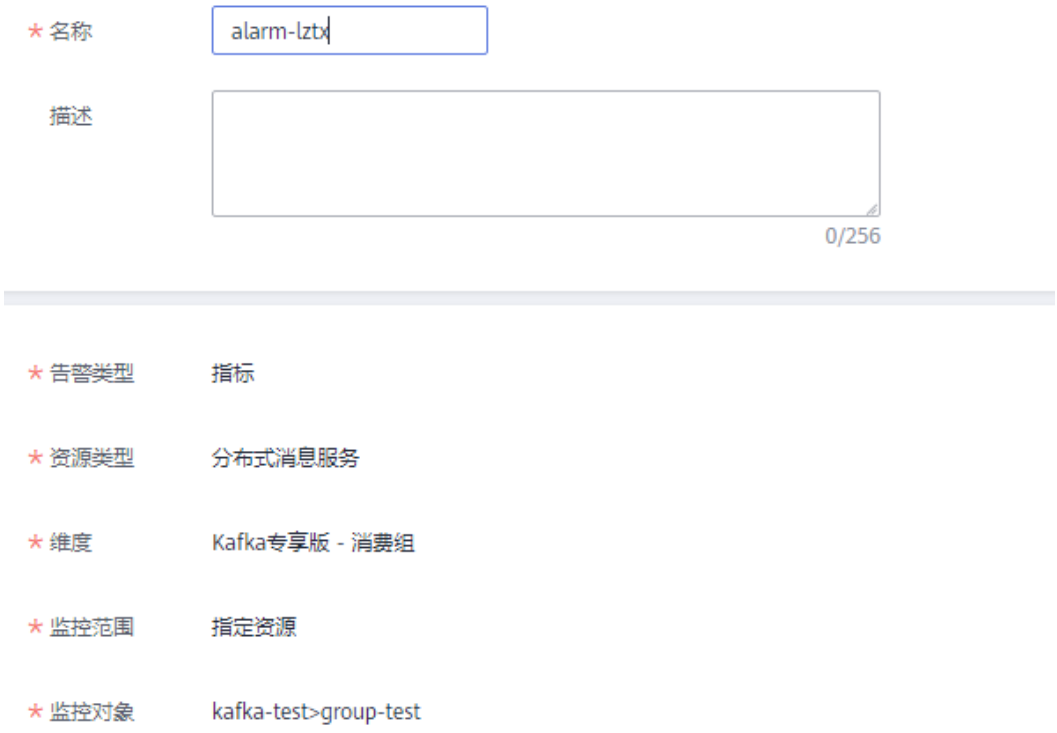
- 消费组：选择需要创建告警通知的消费组。
- 主题：选择“全部Topic”。

步骤4 选中“消息堆积数（消费组可消费消息数）”图表，单击  ，创建告警规则。

图 7-2 消息堆积数图表

步骤5 在“创建告警规则”界面，设置告警名称。

图 7-3 设置告警名称



- 名称：您自定义的告警名称，用于识别不同的告警。
- 描述：告警规则描述，可以不填。

步骤6 在“创建告警规则”界面，设置告警策略。

图 7-4 设置告警策略

- 触发规则：选择“自定义创建”。
- 告警策略：触发告警规则的告警策略，是否触发告警取决于连续周期的数据是否达到阈值。
- 告警级别：根据实际情况选择告警等级。

步骤7 在“创建告警规则”界面，设置告警通知对象。

图 7-5 设置告警通知对象

- 发送通知：选择开启。
- 通知方式：选择“主题订阅”。
- 通知对象：选择云账号联系人或已创建的告警通知主题，告警通知主题的订阅信息中包含需要接收告警信息的手机号/邮箱地址。

  如果尚未创建告警通知主题，参考如下操作创建告警通知主题：单击“创建主题”，进入消息通知服务中，**创建主题**和**添加订阅**。创建完成后，返回“创建告警规则”页面，在“通知对象”后单击，然后选择创建的告警通知主题。

  说明

  在添加订阅后，对应的订阅终端会收到订阅通知，用户要选择确认订阅，后续才能收到告警信息。

图 7-6 创建告警通知主题

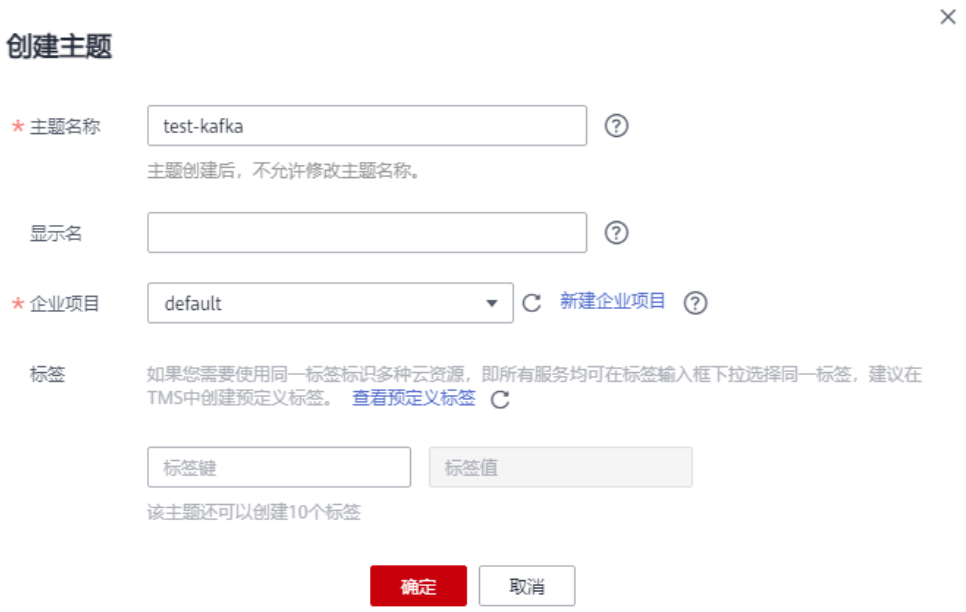


图 7-7 添加订阅

- 生效时间：该告警规则仅在生效时间内发送通知消息。
- 触发条件：触发告警通知的条件。

步骤8 在“创建告警规则”界面，设置企业项目和标签。

图 7-8 设置企业项目和标签

- 归属企业项目：告警规则所属的企业项目。只有拥有该企业项目权限的用户才可以查看和管理该告警规则。
- 标签：标签用于标识云资源，当您拥有相同类型的许多云资源时，可以使用标签按各种维度（例如用途、所有者或环境）对云资源进行分类。

步骤9 单击“立即创建”，完成告警规则的设置。

告警规则创建完成后，在云监控服务的“告警 > 告警规则”界面，查看新创建的告警规则。

图 7-9 查看新创建的告警规则

----结束

8 Logstash 对接 Kafka

应用场景

Logstash是免费且开放的服务器端数据处理管道，能够从多个来源采集数据，转换数据，然后将数据发送到指定的存储中。Kafka是一种高吞吐量的分布式发布订阅消息系统，也是Logstash支持的众多输入输出源之一。本章节主要介绍Logstash如何对接Kafka实例。

方案架构

- Kafka实例作为Logstash输入源的示意图如下。

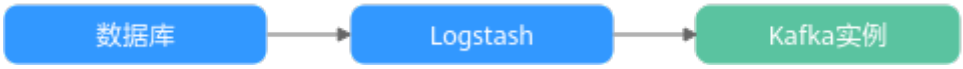
  图 8-1 Kafka 实例作为 Logstash 输入源

  日志采集客户端将数据发送到Kafka实例中，Logstash根据自身性能从Kafka实例中拉取数据。Kafka实例作为Logstash输入源时，可以防止突发流量对于Logstash的影响，以及解耦日志采集客户端和Logstash，保证系统的稳定性。

- Kafka实例作为Logstash输出源的示意图如下。

  图 8-2 Kafka 实例作为 Logstash 输出源

  Logstash从数据库采集数据，然后发送到Kafka实例中进行存储。Kafka实例作为Logstash输出源时，由于Kafka的高吞吐量，可以存储大量数据。

约束与限制

Logstash从7.5版本开始支持Kafka Integration Plugin插件，Kafka Integration Plugin插件包含Kafka input Plugin和Kafka output Plugin。Kafka input Plugin用于从Kafka实例的Topic中读取数据，Kafka output Plugin把数据写入到Kafka实例的Topic。Logstash、Kafka Integration Plugin与Kafka客户端的版本对应关系如表8-1所示。**请确保Kafka客户端版本大于或等于Kafka实例的版本**。

表 8-1 版本对应关系

Logstash版本	Kafka Integration Plugin版本	Kafka客户端版本
8.3~8.8	10.12.0	2.8.1
8.0~8.2	10.9.0~10.10.0	2.5.1
7.12~7.17	10.7.4~10.9.0	2.5.1
7.8~7.11	10.2.0~10.7.1	2.4
7.6~7.7	10.0.1	2.3.0
7.5	10.0.0	2.1.0

前提条件

执行实施步骤前，请确保已完成以下操作：

- 下载Logstash。
- 准备一台Windows系统的主机，在主机中安装Java Development Kit 1.8.111或以上版本和Git Bash。
- 创建Kafka实例和Topic，并获取Kafka实例信息。

  Kafka实例未开启公网访问和SASL认证时，获取表8-2所示信息。

  表 8-2 Kafka 实例信息（未开启公网访问和 SASL 认证）

参数名	获取途径
内网连接地址	在Kafka实例详情页的“连接信息”区域，获取“内网连接地址”。
Topic名称	在Kafka实例控制台，单击实例名称，进入实例详情页。在左侧导航栏单击“Topic管理”，进入Topic列表页面，获取Topic名称。 下文以topic-logstash为例介绍。

  Kafka实例未开启公网访问、已开启SASL认证时，获取表8-3所示信息。

  表 8-3 Kafka 实例信息（未开启公网访问、已开启 SASL 认证）

参数名	获取途径
内网连接地址	在Kafka实例详情页的“连接信息”区域，获取“内网连接地址”。
开启的SASL认证机制	在Kafka实例详情页的“连接信息”区域，获取“开启的SASL认证机制”。

参数名	获取途径
启用的安全协议	在Kafka实例详情页的“连接信息”区域，获取“启用的安全协议”。
证书	在Kafka实例详情页的“连接信息”区域，在“SSL证书”所在行，单击“下载”。下载压缩包后解压，获取压缩包中的客户端证书文件：client.truststore.jks。
SASL用户名和密码	在Kafka实例控制台，单击实例名称，进入实例详情页。在左侧导航栏单击“用户管理”，进入用户列表页面，获取用户名。如果忘记了密码，单击“重置密码”，重新设置密码。
Topic名称	在Kafka实例控制台，单击实例名称，进入实例详情页。在左侧导航栏单击“Topic管理”，进入Topic列表页面，获取Topic名称。 下文以topic-logstash为例介绍。

Kafka实例已开启公网访问、未开启SASL认证时，获取表8-4所示信息。

表 8-4 Kafka 实例信息（已开启公网访问、未开启 SASL 认证）

参数名	获取途径
公网连接地址	在Kafka实例详情页的“连接信息”区域，获取“公网连接地址”。
Topic名称	在Kafka实例控制台，单击实例名称，进入实例详情页。在左侧导航栏单击“Topic管理”，进入Topic列表页面，获取Topic名称。 下文以topic-logstash为例介绍。

Kafka实例已开启公网访问和SASL认证时，获取表8-5所示信息。

表 8-5 Kafka 实例信息（已开启公网访问和 SASL 认证）

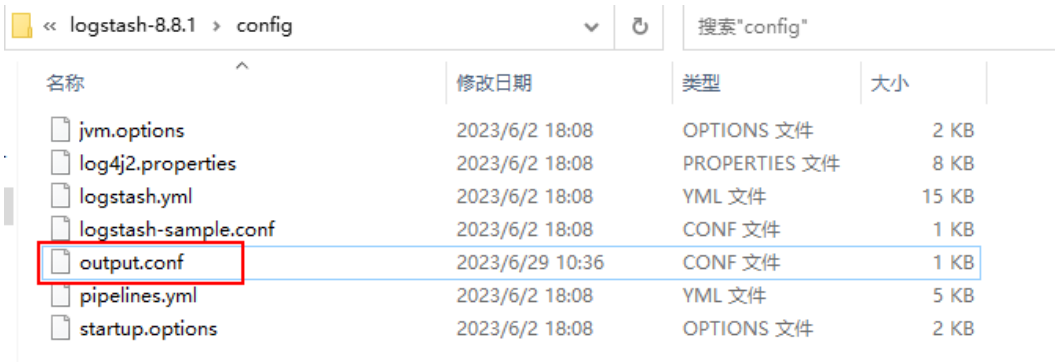
参数名	获取途径
公网连接地址	在实例详情页的“连接信息”区域，获取“公网连接地址”
开启的SASL认证机制	在Kafka实例详情页的“连接信息”区域，获取“开启的SASL认证机制”。
启用的安全协议	在Kafka实例详情页的“连接信息”区域，获取“启用的安全协议”。
证书	在Kafka实例详情页的“连接信息”区域，在“SSL证书”所在行，单击“下载”。下载压缩包后解压，获取压缩包中的客户端证书文件：client.truststore.jks。

参数名	获取途径
SASL用户名和密码	在Kafka实例控制台，单击实例名称，进入实例详情页。在左侧导航栏单击“用户管理”，进入用户列表页面，获取用户名。如果忘记了密码，单击“重置密码”，重新设置密码。
Topic名称	在Kafka实例控制台，单击实例名称，进入实例详情页。在左侧导航栏单击“Topic管理”，进入Topic列表页面，获取Topic名称。 下文以topic-logstash为例介绍。

实施步骤（Kafka 实例作为 Logstash 输出源）

步骤1 在Windows主机中，解压Logstash压缩包，进入“config”文件夹，创建“output.conf”配置文件。

图 8-3 创建“output.conf”配置文件



步骤2 在“output.conf”配置文件中，增加如下内容，连接Kafka实例。

```
input {
    stdin {}
}
output {
 kafka {
    bootstrap_servers => "ip1:port1,ip2:port2,ip3:port3"
    topic_id => "topic-logstash"

    #如果不使用SASL认证，以下参数请注释掉。
    #SASL认证机制为“PLAIN”时，配置信息如下。
    sasl_mechanism => "PLAIN"
    sasl_jaas_config => "org.apache.kafka.common.security.plain.PlainLoginModule required
username='username' password='password';"

    #SASL认证机制为“SCRAM-SHA-512”时，配置信息如下。
    sasl_mechanism => "SCRAM-SHA-512"
    sasl_jaas_config => "org.apache.kafka.common.security.scram.ScramLoginModule required
username='username' password='password';"

    #安全协议为“SASL_SSL”时，配置信息如下。
    security_protocol => "SASL_SSL"
    ssl_truststore_location => "C:\\Users\\Desktop\\logstash-8.8.1\\config\\client.jks"
    ssl_truststore_password => "dms@kafka"
    ssl_endpoint_identification_algorithm => ""

    #安全协议为“SASL_PLAINTEXT”时，配置信息如下。
    security_protocol => "SASL_PLAINTEXT"
    }
}
```

参数说明如下：

- bootstrap_servers：Kafka实例的“内网连接地址”/“公网连接地址”
- topics：Topic名称
- sasl_mechanism：SASL认证机制
- sasl_jaas_config：SASL jaas的配置文件，根据实际情况修改SASL用户名和密码
- security_protocol：Kafka实例的安全协议
- ssl_truststore_location：SSL证书的存放位置
- ssl_truststore_password：服务器证书密码，**不可更改，需要保持为dms@kafka**。
- ssl_endpoint_identification_algorithm：证书域名校验开关，为空则表示关闭。**这里需要保持关闭状态，必须设置为空**。

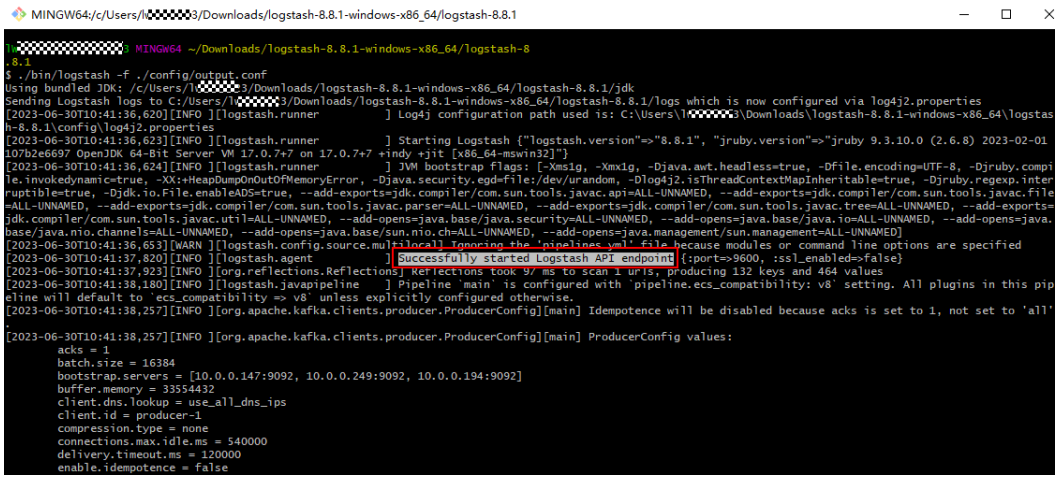
如果需要了解Kafka output Plugin的其他参数，请参见**Kafka output Plugin**。

步骤3 在Logstash文件夹根目录打开Git Bash，执行以下命令启动Logstash。

```
./bin/logstash -f ./config/output.conf
```

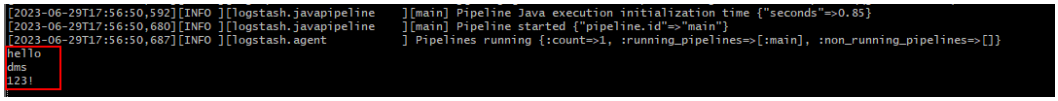
返回“Successfully started Logstash API endpoint”时，表示启动成功。

图 8-4 启动 Logstash



步骤4 在Logstash中，生产消息，如下图所示。

图 8-5 生产消息



步骤5 切换到Kafka控制台，单击实例名称，进入实例详情页。

步骤6 在左侧导航栏单击“消息查询”，进入消息查询页面。

步骤7 在“Topic名称”中选择“topic-logstash”，单击“搜索”，查询消息。

图 8-6 查询消息

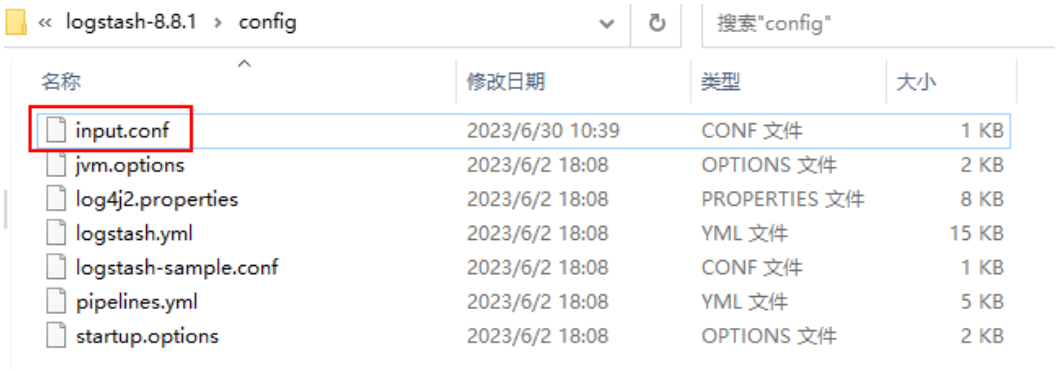
从图8-6可以看出，Logstash的Kafka output Plugin已经把数据写入到Kafka实例的topic-logstash中。

----结束

实施步骤（Kafka 实例作为 Logstash 输入源）

步骤1 在Windows主机中，解压Logstash压缩包，进入“config”文件夹，创建“input.conf”配置文件。

图 8-7 创建“input.conf”配置文件



步骤2 在“input.conf”配置文件中，增加如下内容，连接Kafka实例。

```
input {
 kafka {
    bootstrap_servers => "ip1:port1,ip2:port2,ip3:port3"
    group_id => "logstash_group"
    topic_id => "topic-logstash"
    auto_offset_reset => "earliest"

    #如果不使用SASL认证，以下参数请注释掉。
    #SASL认证机制为“PLAIN”时，配置信息如下。
    sasl_mechanism => "PLAIN"
    sasl_jaas_config => "org.apache.kafka.common.security.plain.PlainLoginModule required
username='username' password='password';"

    #SASL认证机制为“SCRAM-SHA-512”时，配置信息如下。
    sasl_mechanism => "SCRAM-SHA-512"
    sasl_jaas_config => "org.apache.kafka.common.security.scram.ScramLoginModule required
username='username' password='password';"

    #安全协议为“SASL_SSL”时，配置信息如下。
    security_protocol => "SASL_SSL"
    ssl_truststore_location => "C:\\Users\\Desktop\\logstash-8.8.1\\config\\client.jks"
    ssl_truststore_password => "dms@kafka"
    ssl_endpoint_identification_algorithm => ""

    #安全协议为“SASL_PLAINTEXT”时，配置信息如下。
    security_protocol => "SASL_PLAINTEXT"
```

```
    }
}
output {
 stdout{codec=>rubydebug}
}
```

参数说明如下：

- bootstrap_servers：Kafka实例的“内网连接地址”/“公网连接地址”
- group_id：消费组的名称
- topics：Topic名称
- auto_offset_reset：指定消费者的消费策略，本文以earliest为例
- sasl_mechanism：SASL认证机制
- sasl_jaas_config：SASL jaas的配置文件，根据实际情况修改SASL用户名和密码
- security_protocol：Kafka实例的安全协议
- ssl_truststore_location：SSL证书的存放位置
- ssl_truststore_password：服务器证书密码，**不可更改，需要保持为dms@kafka**。
- ssl_endpoint_identification_algorithm：证书域名校验开关，为空则表示关闭。**这里需要保持关闭状态，必须设置为空**。

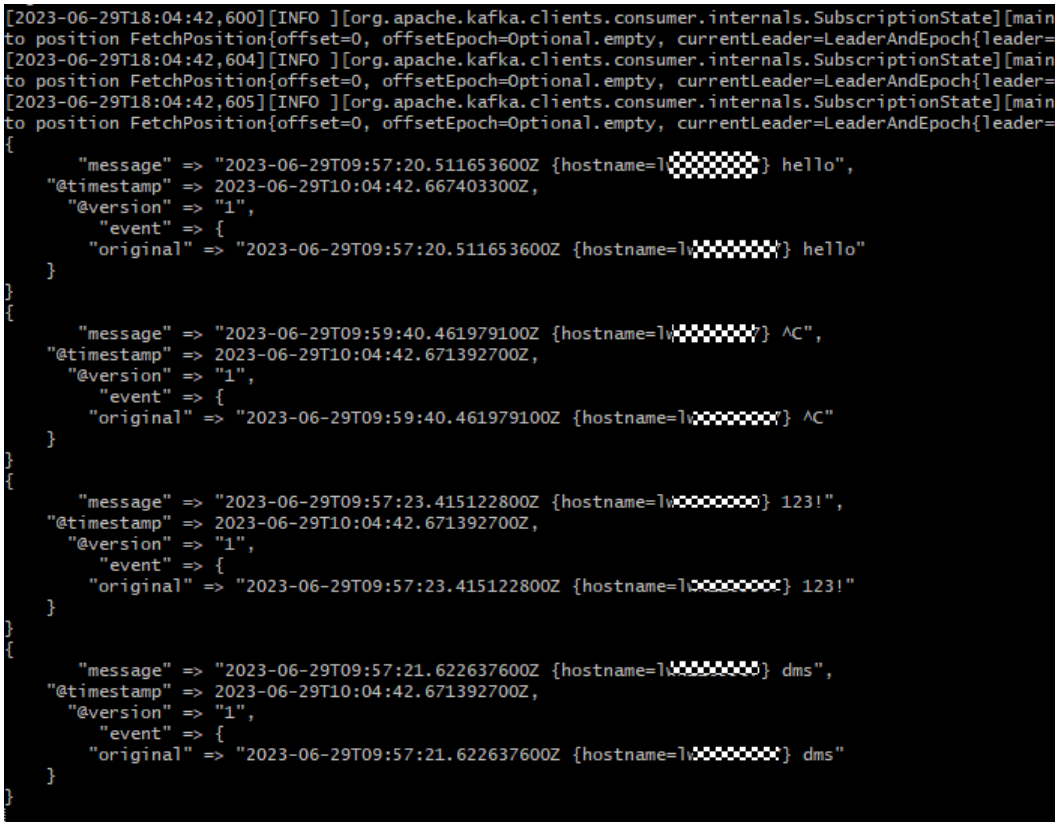
如果需要了解Kafka input Plugin的其他参数，请参见**Kafka input Plugin**。

步骤3 在Logstash文件夹根目录打开Git Bash，执行以下命令启动Logstash。

```
./bin/logstash -f ./config/input.conf
```

Logstash启动成功后，Kafka input Plugin会自动从Kafka实例的topic-logstash中读取数据，如下图所示。

图 8-8 Logstash 从 topic-logstash 中读取的数据

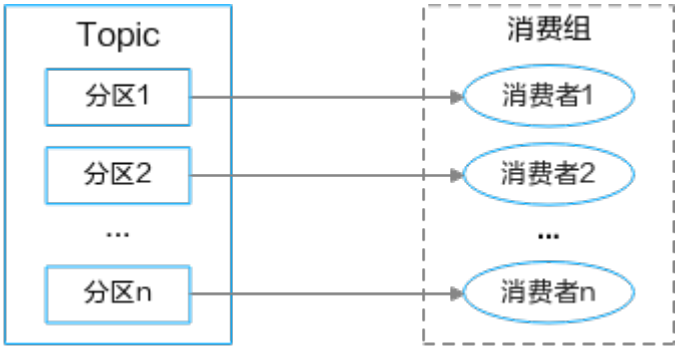


----结束

9 消息堆积最佳实践

方案概述

Kafka将Topic划分为多个分区，消息被分布式存储在分区中。同一个消费组内，一个消费者可同时消费多个分区，但一个分区在同一时刻只能被一个消费者消费。



在消息处理过程中，如果客户端的消费速度跟不上服务端的发送速度，未处理的消息会越来越多，这部分消息就被称为堆积消息。消息没有被及时消费就会产生消息堆积，从而会造成消息消费延迟。

消息堆积原因

导致消息堆积的常见原因如下：

- 生产者短时间内生产大量消息到Topic，消费者无法及时消费。
- 消费者的消费能力不足（消费者并发低、消息处理时间长），导致消费效率低于生产效率。
- 消费者异常（如消费者故障、消费者网络异常等）导致无法消费消息。
- Topic分区设置不合理，或新增分区无消费者消费。
- Topic频繁重平衡导致消费效率降低。

实施步骤

从消息堆积的原因看，消息堆积问题可以从消费者端、生产者端和服务端三个方面进行处理。

- **消费者端**

  - 根据实际业务需求，合理增加消费者个数（消费并发度），确保分区数/消费者数=整数，建议消费者数和分区数保持一致。
  - 提高消费者的消费速度，通过优化消费者处理逻辑（减少复杂计算、第三方接口调用和读库操作），减少消费时间。
  - 增加消费者每次拉取消息的数量：拉取数据/处理时间 >= 生产速度。
- **生产者端**

  生产消息时，给消息Key加随机后缀，使消息均衡分布到不同分区上。

  说明

  在实际业务场景中，为消息Key加随机后缀，会导致消息全局不保序，需根据实际业务判断是否适合给消息Key加随机后缀。
- **服务端**
  - 合理设置Topic的分区数，在不影响业务处理效率的情况下，调大Topic的分区数量。
  - 当服务端出现消息堆积时，对生产者进行熔断，或将生产者的消息转发到其他Topic。

10 业务过载最佳实践

方案概述

Kafka业务过载，一般表现为CPU使用率高、磁盘写满的现象。

- 当CPU使用率过高时，系统的运行速度会降低，并有加速硬件损坏的风险。
- 当磁盘写满时，相应磁盘上的Kafka日志目录会出现offline问题。此时，该磁盘上的分区副本不可读写，降低了分区的可用性和容错能力。同时由于Leader分区迁移到其他节点，会增加其他节点的负载。

CPU使用率高的原因

- 数据操作相关线程数（num.io.threads、num.network.threads、num.replica.fetchers）过多，导致CPU繁忙。
- 分区设置不合理，所有的生产和消费都集中在某个节点上，导致CPU利用率高。

磁盘写满的原因

- 业务数据增长较快，已有的磁盘空间不能满足业务数据需要。
- 节点内磁盘使用率不均衡，生产的消息集中在某个分区，导致分区所在的磁盘写满。
- Topic的数据老化时间设置过大，保存了过多的历史数据，容易导致磁盘写满。

实施步骤

CPU使用率高的处理措施：

- 优化线程参数num.io.threads、num.network.threads和num.replica.fetchers的配置。
  - num.io.threads和num.network.threads建议配置为磁盘个数的倍数，但不能超过CPU核数。
  - num.replica.fetchers建议配置不超过5。
- 合理设置Topic的分区，分区一般设置为节点个数的倍数。
- 生产消息时，给消息Key加随机后缀，使消息均衡分布到不同分区上。

  说明

  在实际业务场景中，为消息Key加随机后缀，会导致消息全局不保序，需根据实际业务判断是否适合给消息Key加随机后缀。

磁盘写满的处理措施：

- 扩容磁盘，使磁盘具备更大的存储空间。
- 迁移分区，将已写满的磁盘中的分区迁移到本节点的其他磁盘中。
- 合理设置Topic的数据老化时间，减少历史数据的容量大小。
- 在CPU资源情况可控的情况下，使用压缩算法对数据进行压缩。

  常用的压缩算法包括：ZIP，GZIP，SNAPPY，LZ4等。选择压缩算法时，需考虑数据的压缩率和压缩耗时。通常压缩率越高的算法，压缩耗时也越高。对于性能要求高的系统，可以选择压缩速度快的算法，比如LZ4；对于需要更高压缩比的系统，可以选择压缩率高的算法，比如GZIP。

  可以在生产者端配置“compression.type”参数来启用指定类型的压缩算法。

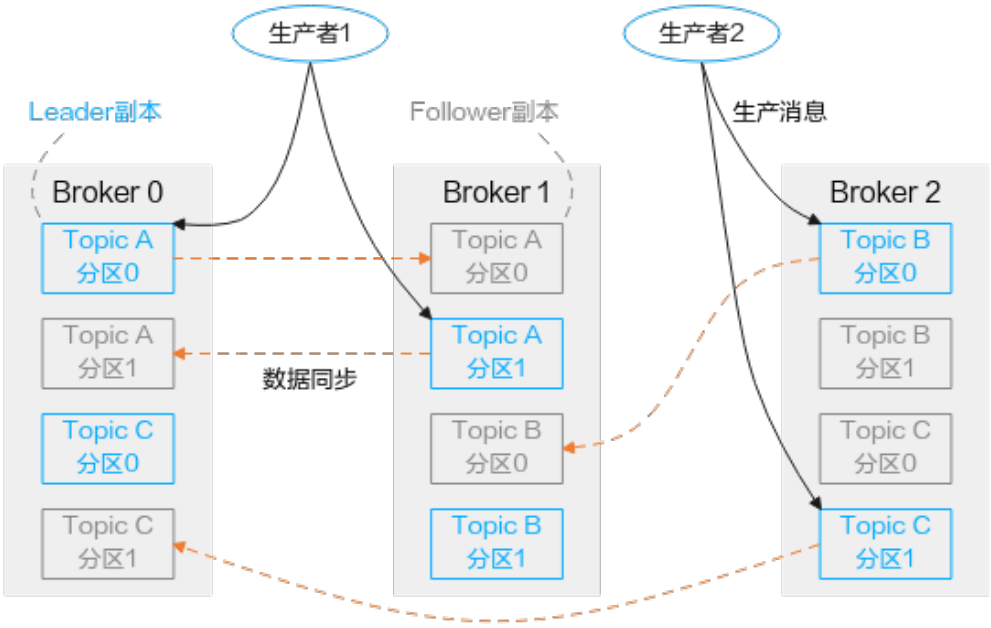
```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("acks", "all");
props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
// 开启GZIP压缩
props.put("compression.type", "gzip");

Producer<String, String> producer = new KafkaProducer<>(props);
```

11 业务数据不均衡最佳实践

方案概述

Kafka将Topic划分为多个分区，所有消息分布式存储在各个分区上。每个分区有一个或多个副本，分布在不同的Broker节点上，每个副本存储一份全量数据，副本之间的消息数据保持同步。Kafka的Topic、分区、副本和代理的关系如下图所示：



在实际业务过程中可能会遇到各节点间或分区之间业务数据不均衡的情况，业务数据不均衡会降低Kafka集群的性能，降低资源使用率。

业务数据不均衡原因

- 业务中部分Topic的流量远大于其他Topic，会导致节点间的数据不均衡。
- 生产者发送消息时指定了分区，未指定的分区没有消息，会导致分区间的数据不均衡。
- 生产者发送消息时指定了消息Key，按照对应的Key发送消息至对应的分区，会导致分区间的数据不均衡。
- 系统重新实现了分区分配策略，但策略逻辑有问题，会导致分区间的数据不均衡。

- Kafka扩容了Broker节点，新增的节点没有分配分区，会导致节点间的数据不均衡。
- 业务使用过程中随着集群状态的变化，多少会发生一些Leader副本的切换或迁移，会导致个别Broker节点上的数据更多，从而导致节点间的数据不均衡。

实施步骤

业务数据不均衡的处理措施：

- 优化业务中Topic的设计，对于数据量特别大的Topic，可对业务数据做进一步的细分，并分配到不同的Topic上。
- 生产者生产消息时，尽量把消息均衡发送到不同的分区上，确保分区间的数据均衡。
- 创建Topic时，使分区的Leader副本分散到各个Broker节点中，以保障整体的数据均衡。
- Kafka提供了分区重平衡的功能，可以把分区的副本重新分配到不同的Broker节点上，解决节点间负载不均衡的问题。具体分区重平衡的操作请参考**修改分区平衡**。

12 Python 调用 SDK 生产消息和查询消息

应用场景

本文介绍以SDK方式调用分布式消息服务Kafka版接口，实现生产消息和查询消息的相关示例。

前提条件

- 使用华为云Python SDK前，您需要拥有华为云账号及该账号对应的AK/SK。获取AK/SK的方法，请参考**访问密钥**。
- 华为云Python SDK支持Python 3.3以上的版本，请确保Python版本满足要求。执行**python --version**，检查当前Python的版本。

实施步骤

步骤1 安装SDK依赖包。

- 使用pip安装
  ```
  pip install huaweicloudsdkkafka
  ```
- 使用源码安装
  ```
  cd huaweicloudsdkkafka-${version}
  python setup.py install
  ```

步骤2 生产消息和查询消息。

AK/SK直接存放在代码中存在安全风险，建议在配置文件或者环境变量中通过密文存放，在使用时解密。

```
import os

from huaweicloudsdkcore.auth.credentials import BasicCredentials
from huaweicloudsdkcore.exceptions import exceptions
from huaweicloudsdkkafka.v2.kafka_client import KafkaClient
from huaweicloudsdkkafka.v2.model.send_kafka_message_req import SendKafkaMessageReq
from huaweicloudsdkkafka.v2.model.send_kafka_message_request import SendKafkaMessageRequest
from huaweicloudsdkkafka.v2.model.show_instance_messages_request import
ShowInstanceMessagesRequest


class KafkaMessageDemo:

    def __init__(self):
        pass
```

```
    @staticmethod
    def main(args):
        # 本示例将AK/SK保存在环境变量中来实现身份验证，运行本示例前请先在本地环境中设置环境变量
HUAWEICLOUD_SDK_AK和HUAWEICLOUD_SDK_SK。
        ak = os.environ["HUAWEICLOUD_SDK_AK"]
        sk = os.environ["HUAWEICLOUD_SDK_SK"]
        endpoint = "<YOUR Endpoint>"
        project_id = "<YOUR ProjectId>"

        credentials = BasicCredentials(
            ak=ak,
            sk=sk,
            project_id=project_id
        )

        client = KafkaClient.new_builder() \
            .with_credentials(credentials) \
            .with_endpoint(endpoint=endpoint) \
            .build()

        # 生产消息
        KafkaMessageDemo.__send_kafka_message(client)
        # 查询消息
        KafkaMessageDemo.__show_instance_messages(client)

    @staticmethod
    def __send_kafka_message(client):
        try:
            body = SendKafkaMessageReq()
            body.topic = "<YOUR Topic>"
            body.body = "<YOUR Body>"
            # property是Topic的分区信息等，如{"name:PARTITION","value:XXXX"}
            propertyList = []
            property = dict()
            property["<YOUR Key>"] = "<YOUR Value>"
            propertyList.append(property)
            body.property_list = propertyList

            request = SendKafkaMessageRequest()
            request.instance_id = "<YOUR InstanceId>"
            # 生产消息对应的action_id为send
            request.action_id = "<YOUR ActionId>"
            request.body = body
            response = client.send_kafka_message(request)
            print(response)
        except exceptions.ClientRequestException as e:
            print(e.status_code)
            print(e.request_id)
            print(e.error_code)
            print(e.error_msg)

    @staticmethod
    def __show_instance_messages(client):
        try:
            request = ShowInstanceMessagesRequest()
            request.instance_id = "<YOUR InstanceId>"
            request.topic = "<YOUR Topic>"
            # asc代表是否按照时间排序
            request.asc = "<YOUR Asc>"
            request.start_time = "<YOUR StartTime>"
            request.end_time = "<YOUR EndTime>"
            request.limit = "<YOUR Limit>"
            request.offset = "<YOUR Offset>"
            # download代表是否下载
            request.download = "<YOUR Download>"
            request.message_offset = "<YOUR MessageOffset>"
            request.partition = "<YOUR Partition>"
            request.keyword = "<YOUR Keyword>"
            response = client.show_instance_messages(request)
```



```
            print(response)
        except exceptions.ClientRequestException as e:
            print(e.status_code)
            print(e.request_id)
            print(e.error_code)
            print(e.error_msg)


if __name__ == "__main__":
    KafkaMessageDemo().main(any)
```

----结束