消息队列 RabbitMQ

消息队列 KafkaMQ

消息队列 RabbitMQ

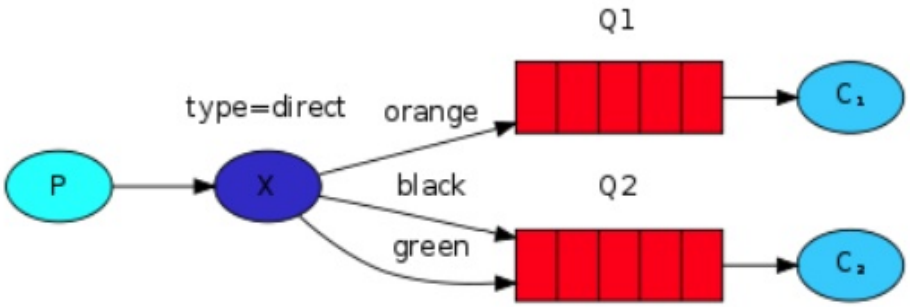
1. RabbitMQ 概述

1.1 RabbitMQ 是什么

- RabbitMQ (Rabbit Message Queue) 是一款开源的消息队列系统
- RabbitMQ 的主要特点在于健壮性好、易于使用、高性能、高并发、集群易扩展，以及强大的开源社区支持

1.2 RabbitMQ 角色

- P 是 Producer，代表生产者，也就是消息的发送者，可以将消息发送到 X
- X 是 Exchange，代表交换机，可以接受生产者发送的消息，并根据路由将消息发送给指定的队列
- Q 是 Queue，也就是队列，存放交换机发送来的消息
- C 是 Consumer，代表消费者，也就是消息的接受者，从队列中获取消息



1.3 环境配置

- 安装 Erlang
  - RabbitMQ 服务器是用 Erlang 语言编写的，它的安装包里并没有集成 Erlang 的环境，因此需要先安装 Erlang
- 安装 RabbitMQ
  - Prompt 输入 rabbitmqctl.bat status 可确认 RabbitMQ 的启动状态
- 输入以下命令来启用客户端管理 UI 插件
  - `rabbitmq-plugins enable rabbitmq_management`

- 浏览器地址栏输入 http://localhost:15672/open in new window 可以进入管理端界面

2. 使用 RabbitMQ

2.1 Java 链接 RabbitMQ

- 在项目中添加 RabbitMQ 客户端依赖：

```
<dependency>
    <groupId>com.rabbitmq</groupId>
    <artifactId>amqp-client</artifactId>
    <version>5.9.0</version>
</dependency>
```

- 模拟场景: 一个生产者发送消息到队列中，一个消费者从队列中读取消息
  - 新建生产者类 Producer

```
public class Producer {
    // QUEUE_NAME 为队列名，也就是说，生产者发送的消息会放到 love 队列中
    private final static String QUEUE_NAME = "love";
    public static void main(String[] args) throws IOException,
TimeoutException {
        ConnectionFactory factory = new ConnectionFactory();
        // 创建服务器连接
        try (Connection connection = factory.newConnection();
             Channel channel = connection.createChannel()) {

            // 在发送消息的时候，必须设置队列名称，通过 queueDeclare() 方法设置
            channel.queueDeclare(QUEUE_NAME, false, false, false, null);
            String message = "小巷，我喜欢你。";
            // basicPublish() 方法用于发布消息
            channel.basicPublish("", QUEUE_NAME, null,
message.getBytes(StandardCharsets.UTF_8));
            System.out.println(" [Producer] 发送 '" + message + "'");
        }
    }
}
```

  - ConnectionFactory 是一个非常方便的工厂类，可用来创建到 RabbitMQ 的默认连接（主机名为“localhost”）。然后，创建一个通道（ Channel）来发送消息
  - Connection 和 Channel 类都实现了 Closeable 接口，所以可以使用 try-with-resource 语句
  - basicPublish() 参数
    - 第一个参数为交换机（exchange），当前场景不需要，因此设置为空字符串；
    - 第二个参数为路由关键字（routingKey），暂时使用队列名填充；
    - 第三个参数为消息的其他参数（BasicProperties），暂时不配置；
    - 第四个参数为消息的主体，这里为 UTF-8 格式的字节数组，可以有效地杜绝中文乱码
  - 新建消费者类 Consumer

```
public class Consumer {
    private final static String QUEUE_NAME = "love";
    public static void main(String[] args) throws IOException,
TimeoutException {
        ConnectionFactory factory = new ConnectionFactory();
        Connection connection = factory.newConnection();
        Channel channel = connection.createChannel();

        // 在接收消息的时候，必须设置队列名称，通过 queueDeclare() 方法设置
        channel.queueDeclare(QUEUE_NAME, false, false, false, null);
        System.out.println("等待接收消息");

        // 由于 RabbitMQ 将会通过异步的方式向我们推送消息，因此我们需要提供了一个回调
        // 该回调将对消息进行缓冲，直到我们做好准备接收它们为止
        DeliverCallback deliverCallback = (consumerTag, delivery) -> {
            String message = new String(delivery.getBody(), "UTF-8");
            System.out.println(" [小巷] 接收到的消息 '" + message + "'");
        };
        // basicConsume() 方法用于接收消息
        channel.basicConsume(QUEUE_NAME, true, deliverCallback, consumerTag
-> { });
    }
}
```

- 创建通道的代码和生产者差不多，只不过没有使用 try-with-resource 语句来自动关闭连接和通道，因为我们希望消费者能够一直保持连接，直到我们强制关闭它
- basicConsume() 参数
  - 第一个参数为队列名（queue），和生产者相匹配（love）
  - 第二个参数为 autoAck，如果为 true 的话，表明服务器要一次性交付消息。怎么理解这个概念呢？小伙伴们可以在运行消费者类 XiaoXiang 类之前，先多次运行生产者类 Wanger，向队列中发送多个消息，等到消费者类启动后，你就会看到多条消息一次性接收到了，就像下面这样
  - 第三个参数为 DeliverCallback，也就是消息的回调函数
  - 第四个参数为 CancelCallback

消息队列 KafkaMQ

1. Kafka 概述

1.1 Kafka定义

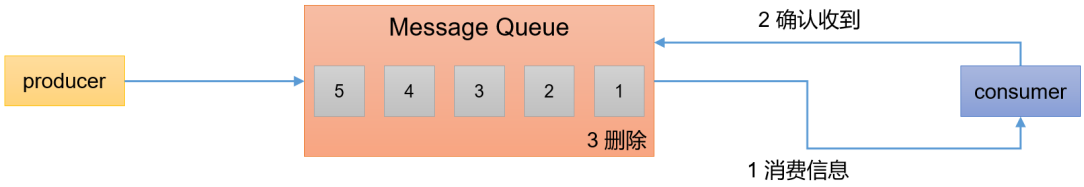
- Kafka传统定义：Kafka是一个分布式的基于发布/订阅模式的消息队列（Message Queue），主要应用于大数据实时处理领域
- 发布/订阅：消息的发布者不会将消息直接发送给特定的订阅者，而是将发布的消息分为不同的类别，订阅者只接收感兴趣的消息
- Kafka最新定义：Kafka是一个开源的分布式事件流平台（Event Streaming Platform），被数千家公司用于高性能数据管道、流分析、数据集成和关键任务应用

1.2 消息队列

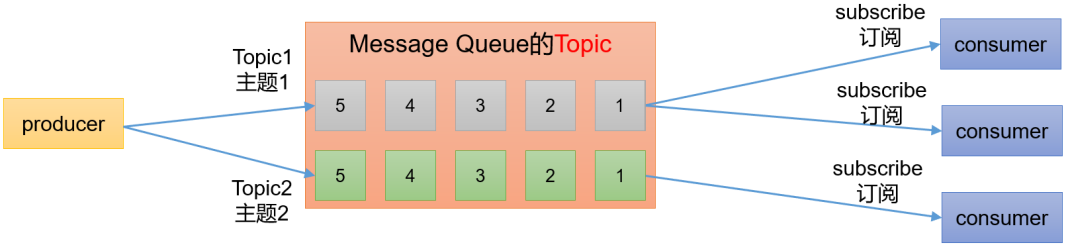
- 目前企业中比较常见的消息队列产品主要有Kafka、ActiveMQ、RabbitMQ、RocketMQ等。
- 在大数据场景主 要采用Kafka作为消息队列; 在JavaEE开发中主要采用ActiveMQ、RabbitMQ、RocketMQ
- 传统消息队列的应用场景
  - 传统的消息队列的主要应用场景包括：缓存/消峰、解耦和异步通信
  - 缓冲/消峰：有助于控制和优化数据流经过系统的速度，解决生产消息和消费消息的处理速度不一致的情况
  - 解耦：允许你独立的扩展或修改两边的处理过程，只要确保它们遵守同样的接口约束(看成生产者-超市-消费者)
  - 异步通信：允许用户把一个消息放入队列，但并不立即处理它，然后在需要的时候再去处理它们

1.3 消息队列的两种模式

- 点对点模式 基于队列的P2P
  - 消息生产者发送消息到队列，消息消费者从队列中接收消息



- 发布/订阅模式 Pub/Sub
  - 可以有多个topic主题（浏览、点赞、收藏、评论等）
  - 消费者消费数据之后，不删除数据
  - 每个消费者相互独立，都可以消费到数据

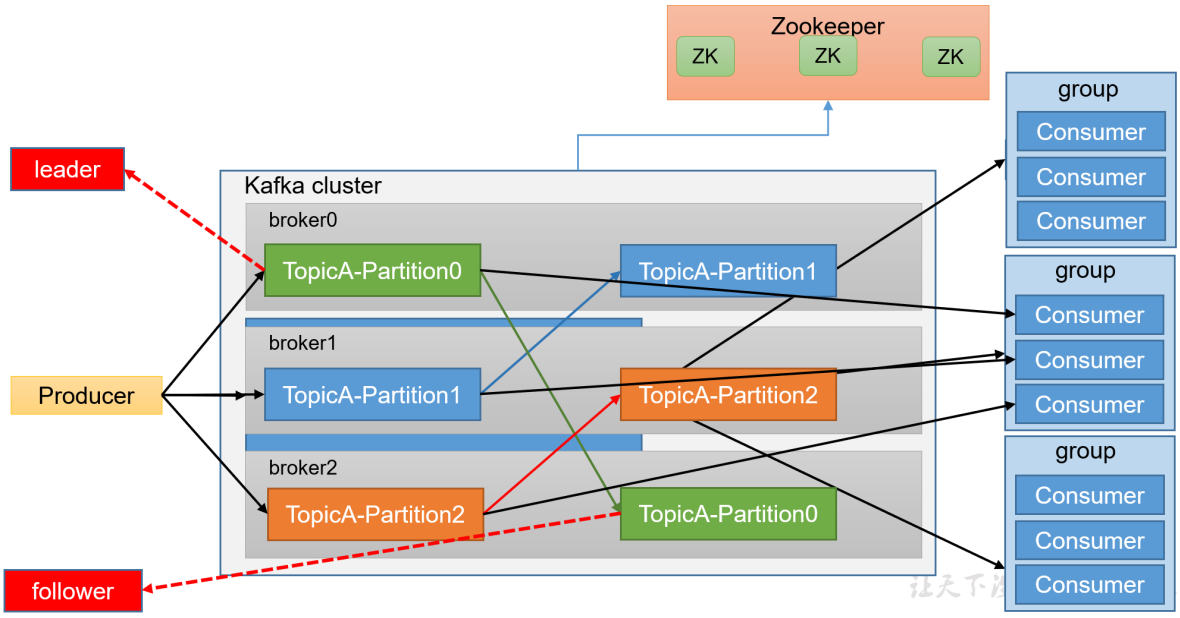


- 消费者与消费组模型的契合：

- 如果所有的消费者都隶属于同一个消费组，那么所有的消息都会被均衡地投递给每一个消费者，即每条消息只会被一个消费者处理，这就相当于点对点模式的应用
- 如果所有的消费者都隶属于不同的消费组，那么所有的消息都会被广播给所有的消费者，即每条消息会被所有的消费者处理，这就相当于发布/订阅模式的应用

1.4 Kafka基础架构

- 为方便扩展，并提高吞吐量，一个topic分为多个partition
- 配合分区的设计，提出消费者组的概念，组内每个消费者并行消费
- 为提高可用性，为每个partition增加若干副本, 副本分Leader/Follower，只有Leader能生产消费
- ZooKeeper中记录哪个broker正在工作，以及谁是leader，Kafka2.8.0以后也可以配置不采用ZK



- Producer：消息生产者，就是向Kafka broker发消息的客户端
- Consumer：消息消费者，向Kafka broker取消息的客户端
- Consumer Group(CG)：消费者组，由多个consumer组成. 消费者组内每个消费者负责消费不同分区的数据，一个分区只能由一个组内消费者消费；消费者组之间互不影响。所有的消费者都属于某个消费者组，即消费者组是逻辑上的一个订阅者
- Broker：一台Kafka服务器就是一个broker。一个集群由多个broker组成。一个broker可以容纳多个topic
- Topic：可以理解为一个队列，生产者和消费者面向的都是一个topic
- Partition：为了实现扩展性，一个非常大的topic可以分布到多个broker（即服务器）上，一个topic可以分为多个partition，每个partition是一个有序的队列
- Replica：副本。一个topic的每个分区都有若干个副本，一个Leader和若干个Follower
- Leader：每个分区多个副本的“主”，生产者发送数据的对象，以及消费者消费数据的对象都是Leader
- Follower：每个分区多个副本中的“从”，实时从Leader中同步数据，保持和Leader数据的同步。Leader发生故障时，某个
- Follower会成为新的Leader

2. Kafka 入门

2.1 Kafka 集群操作

- 集群启停脚本

```
#! /bin/bash

case $1 in
"start"){
    for i in hadoop102 hadoop103 hadoop104
    do
        echo " --------启动 $i Kafka-------"
        ssh $i "/opt/module/kafka/bin/kafka-server-start.sh -daemon
/opt/module/kafka/config/server.properties"
    done
};;
"stop"){
    for i in hadoop102 hadoop103 hadoop104
    do
        echo " --------停止 $i Kafka-------"
        ssh $i "/opt/module/kafka/bin/kafka-server-stop.sh "
    done
};;
esac
```

- 启动集群命令 kf.sh start
- 停止集群命令 kf.sh stop
- 注意：停止Kafka集群时，一定要等Kafka所有节点进程全部停止后再停止Zookeeper集群。因为Zookeeper集群当中记录着Kafka集群相关信息，Zookeeper集群一旦先停止，Kafka集群就没有办法再获取停止进程的信息，只能手动杀死Kafka进程了

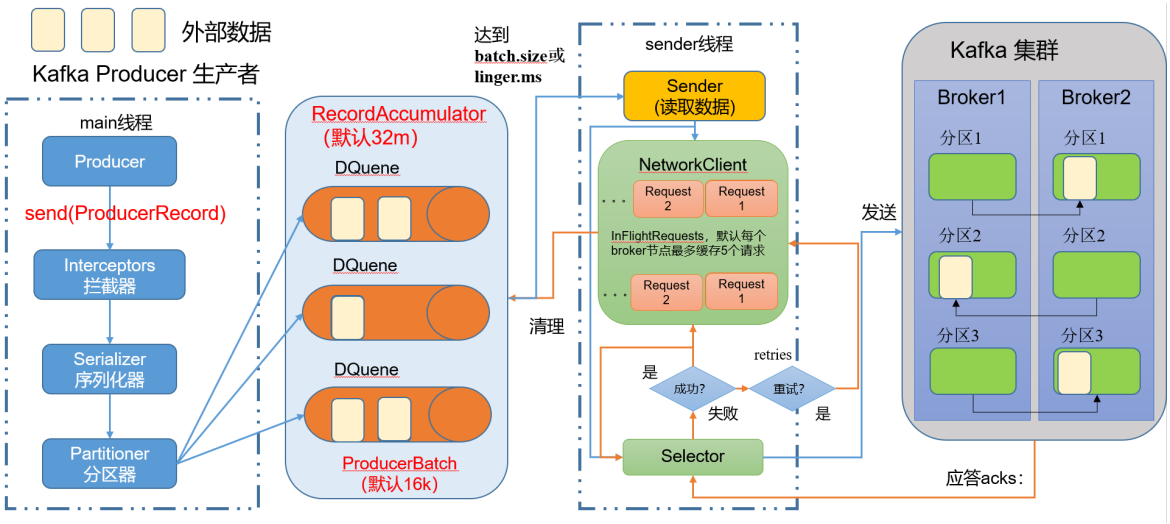
2.2 Kafka命令行操作

- Kafka分为Kafka生产者，消费者，和Kafka集群，对每一个模块有针对的脚本
- 主题命令行操作 kafka-topics.sh
  - 查看操作主题命令参数 `bin/kafka-topics.sh`
  - 查看当前服务器中的所有topic `bin/kafka-topics.sh --bootstrap-server hadoop102:9092 --list`
- 生产者命令行操作 kafka-console-producer.sh
  - 查看操作生产者命令参数 `bin/kafka-console-producer.sh`
- 消费者命令行操作 kafka-console-consumer.sh
  - 查看操作消费者命令参数 `bin/kafka-console-consumer.sh`

3. Kafka 生产者

3.1 发送原理

- 在消息发送的过程中，涉及到了两个线程——main线程和Sender线程
- 在main线程中创建了一个双端队列RecordAccumulator
- main线程将消息发送给RecordAccumulator，Sender线程不断从RecordAccumulator中拉取消息发送到Kafka Broker



  - batch.size：只有数据积累到batch.size之后，sender才会发送数据。默认16k
  - linger.ms：如果数据迟迟未达到batch.size，sender等待linger.ms设置的时间到了之后就会发送数据。单位ms，默认值是0ms，表示没有延迟

- 应答acks：
  - 0：生产者发送过来的数据，不需要等数据落盘应答
  - 1：生产者发送过来的数据，Leader收到数据后应答
  - -1（all）：生产者发送过来的数据，Leader和ISR队列里面的所有节点收齐数据后应答。-1和all等价。

3.2 生产者重要参数列表

参数名称	描述
bootstrap.servers	生产者连接集群所需的broker地址清单。例如hadoop102:9092,hadoop103:9092,hadoop104:9092，可以设置1个或者多个，中间用逗号隔开。注意这里并非需要所有的broker地址，因为生产者从给定的broker里查找到其他broker信息。
key.serializer和value.serializer	指定发送消息的key和value的序列化类型。一定要写全类名。
buffer.memory	RecordAccumulator缓冲区总大小，默认32m。
batch.size	缓冲区一批数据最大值，默认16k。适当增加该值，可以提高吞吐量，但是如果该值设置太大，会导致数据传输延迟增加。
linger.ms	如果数据迟迟未达到batch.size，sender等待linger.time之后就会发送数据。单位ms，默认值是0ms，表示没有延迟。生产环境建议该值大小为5-100ms之间。
acks	0：生产者发送过来的数据，不需要等数据落盘应答。1：生产者发送过来的数据，Leader收到数据后应答。-1（all）：生产者发送过来的数据，Leader+和isr队列里面的所有节点收齐数据后应答。默认值是-1，-1和all是等价的。
max.in.flight.requests.per.connection	允许最多没有返回ack的次数，默认为5，开启幂等性要保证该值是 1-5的数字。
retries	当消息发送出现错误的时候，系统会重发消息。retries表示重试次数。默认是int最大值，2147483647。如果设置了重试，还想保证消息的有序性，需要设置MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION=1否则在重试此失败消息的时候，其他的消息可能发送成功了。
retry.backoff.ms	两次重试之间的时间间隔，默认是100ms。
enable.idempotence	是否开启幂等性，默认true，开启幂等性。
compression.type	生产者发送的所有数据的压缩方式。默认是none，也就是不压缩。 支持压缩类型：none、gzip、snappy、lz4和zstd。

3.3 异步发送API

- 普通异步发送
  - 需求：创建Kafka生产者，采用异步的方式发送到Kafka Broker
  - 创建工程kafka,导入依赖

```
<dependencies>
     <dependency>
          <groupId>org.apache.kafka</groupId>
          <artifactId>kafka-clients</artifactId>
          <version>3.0.0</version>
     </dependency>
</dependencies>
```

  - 编写不带回调函数的API代码 send(ProducerRecord)

```
package com.atguigu.kafka.producer;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerRecord;
import java.util.Properties;

public class CustomProducer {
    public static void main(String[] args) throws InterruptedException {
        // 1. 创建kafka生产者的配置对象
        Properties properties = new Properties();
        // 2. 给kafka配置对象添加配置信息：bootstrap.servers
        properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
"hadoop102:9092");
        // key,value序列化（必须）：key.serializer，value.serializer
        properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
                       StringSerializer.class.getName());
        properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
                       StringSerializer.class.getName());
        // 3. 创建kafka生产者对象
        KafkaProducer<String, String> kafkaProducer = new
KafkaProducer<String, String>(properties);
        // 4. 调用send方法,发送消息
        for (int i = 0; i < 5; i++) {
            kafkaProducer.send(new ProducerRecord<>("first","atguigu " +
i));
        }
        // 5. 关闭资源
        kafkaProducer.close();
}
```

- 带回调函数的异步发送
  - 回调函数会在producer收到ack时调用，为异步调用，该方法有两个参数，分别是元数据信息(RecordMetadata)和异常信息(Exception)，如果Exception为null，说明消息发送成功，如果Exception不为null，说明消息发送失败

- 注意：消息发送失败会自动重试，不需要我们在回调函数中手动重试
- send(ProducerRecord,Callback)

```
package com.atguigu.kafka.producer;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerRecord;
import java.util.Properties;

public class CustomProducerCallback {
    public static void main(String[] args) throws InterruptedException {
        // 1. 创建kafka生产者的配置对象
        Properties properties = new Properties();
        // 2. 给kafka配置对象添加配置信息：bootstrap.servers
        properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
"hadoop102:9092");
        // key,value序列化（必须）：key.serializer，value.serializer
        properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
                       StringSerializer.class.getName());
        properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
                       StringSerializer.class.getName());
        // 3. 创建kafka生产者对象
        KafkaProducer<String, String> kafkaProducer = new
KafkaProducer<String, String>(properties);
        // 4. 调用send方法,发送消息
        for (int i = 0; i < 5; i++) {
            // 添加回调
            kafkaProducer.send(new ProducerRecord<>("first", "atguigu " +
i), new Callback() {
                // 该方法在Producer收到ack时调用，为异步调用
                @Override
                public void onCompletion(RecordMetadata metadata, Exception
exception) {
                    if (exception == null) {
                        // 没有异常,输出信息到控制台
                        System.out.println("主题：" + metadata.topic() + "->"
 + "分区：" +
                                           metadata.partition());
                    } else {
                        // 出现异常打印
                        exception.printStackTrace();
                    }
                }
            });
            // 延迟一会会看到数据发往不同分区
            Thread.sleep(2);
        }
        // 5. 关闭资源
        kafkaProducer.close();
    }
}
```

3.4 同步发送API

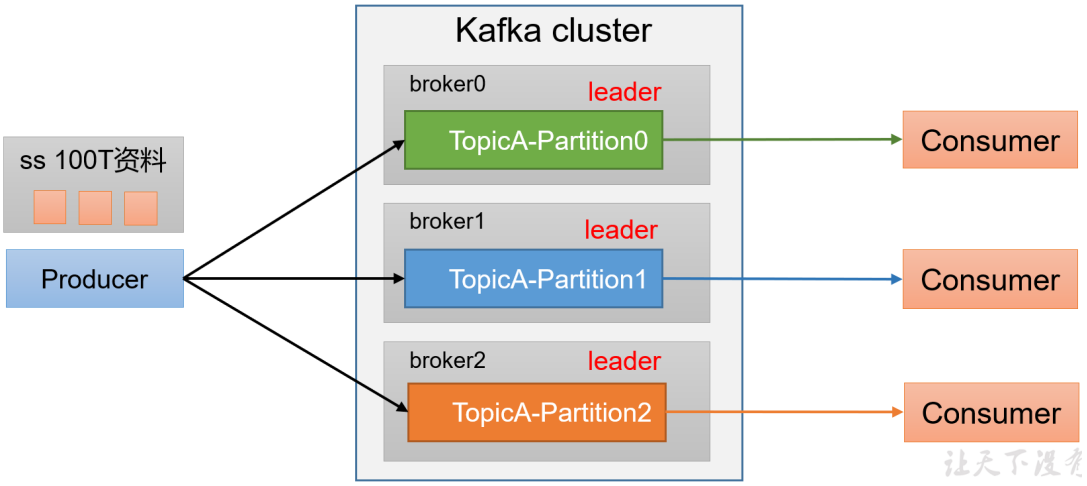
- 只需在异步发送的基础上，再调用一下get()方法即可 send(ProducerRecord)

```
package com.atguigu.kafka.producer;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;
import java.util.Properties;
import java.util.concurrent.ExecutionException;

public class CustomProducerSync{
    public static void main(String[] args) throws InterruptedException,
ExecutionException{
        // 1. 创建kafka生产者的配置对象
        Properties properties = new Properties();
        // 2. 给kafka配置对象添加配置信息

 properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,"hadoop102:9092");
        // key,value序列化（必须）：key.serializer，value.serializer
        properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
                       StringSerializer.class.getName());
        properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
                       StringSerializer.class.getName());
        // 3. 创建kafka生产者对象
        KafkaProducer<String, String> kafkaProducer = new KafkaProducer<String,
String>(properties);
        // 4. 调用send方法,发送消息
        for (int i = 0; i < 10; i++) {
            // 异步发送 默认
            // kafkaProducer.send(new ProducerRecord<>("first","kafka" + i));
            // 同步发送
            kafkaProducer.send(new ProducerRecord<>("first","kafka" + i)).get();
        }
        // 5. 关闭资源
        kafkaProducer.close();
    }
}
```

3.5 生产者分区

- 为什么分区
  - Kafka的消息组织方式实际上是三级结构：主题-分区-消息
  - 主题下的每条消息只会保存在某一个分区中，而不会在多个分区中被保存多份
  - 分区的作用就是提供负载均衡的能力，或者说对数据进行分区的主要原因，就是为了实现系统的高伸缩性（Scalability）
  - 不同的分区能够被放置到不同节点的机器上，而数据的读写操作也都是针对分区这个粒度而进行的，这样每个节点的机器都能独立地执行各自分区的读写请求处理，并且，我们还可以通过添加新的节点机器来增加整体系统的吞吐量

- 分区好处
  - 便于合理使用存储资源，每个Partition在一个Broker上存储，可以把海量的数据按照分区切割成一块一块数据存储在多台Broker上。合理控制分区的任务，可以实现负载均衡的效果
  - 提高并行度，生产者可以以分区为单位发送数据；消费者可以以分区为单位进行消费数据



- 生产者发送消息的分区策略
  - 默认的分区器 DefaultPartitioner
    - 如果记录中指定了分区，则使用指定分区

```
kafkaProducer.send(new ProducerRecord<>("first", 1, "", "atguigu " +
i))
```

    - 如果未指定分区但存在键，则根据键的散列选择分区

```
kafkaProducer.send(new ProducerRecord<>("first", "a", "atguigu " + i))
```

    - 如果不存在分区或密钥，请选择在批次已满时更改的粘性分区
  - 轮询策略
    - 也称Round-robin策略，即顺序分配，轮询策略是Kafka Java生产者API默认提供的分区策略
    - 一个主题下有3个分区，那么第一条消息被发送到分区0，第二条被发送到分区1，第三条被发送到分区2，以此类推。当生产第4条消息时又会重新开始，即将其分配到分区0
    - 轮询策略有非常优秀的负载均衡表现，它总是能保证消息最大限度地被平均分配到所有分区上，故默认情况下它是最合理的分区策略，也是最常用的分区策略之一
  - 随机策略
    - 也称Randomness策略，所谓随机就是随意地将消息放置到任意一个分区上
    - 先计算出该主题总的分区数，然后随机地返回一个小于它的正整数
    - 实现随机策略版的partition方法

```
List partitions = cluster.partitionsForTopic(topic);
return ThreadLocalRandom.current().nextInt(partitions.size());
```

- 按消息键保序策略
  - Kafka允许为每条消息定义消息键，简称为Key，这个Key的作用非常大，它可以是一个有着明确业务含义的字符串，比如客户代码、部门编号或是业务ID等；也可以用来表征消息元数据
  - 一旦消息被定义了Key，那么你就可以保证同一个Key的所有消息都进入到相同的分区里面，由于每个分区下的消息处理都是有顺序的，故这个策略被称为按消息键保序策略
  - 实现这个策略的partition方法

```
List partitions = cluster.partitionsForTopic(topic);
return Math.abs(key.hashCode()) % partitions.size();
```

- 其他分区策略
  - 根据Broker所在的IP地址实现定制化的分区策略

```
List partitions = cluster.partitionsForTopic(topic);
return partitions.stream().filter(p ->
isSouth(p.leader().host())).map(PartitionInfo::partition).findAny().get
();
```

- 自定义分区器
  - 如果要自定义分区策略，需要定义类实现Partitioner接口
  - 重写partition()和close()
  - partition() 方法签名

```
/**
* 返回信息对应的分区
* @param topic          主题
* @param key            消息的key
* @param keyBytes       消息的key序列化后的字节数组
* @param value          消息的value
* @param valueBytes     消息的value序列化后的字节数组
* @param cluster        集群元数据可以查看分区信息
* @return */
int partition(String topic,
              Object key, byte[] keyBytes,
              Object value, byte[] valueBytes, Cluster cluster);
```

  - 需要充分地利用partition参数对消息进行分区，计算出它要被发送到哪个分区

```
package com.atguigu.kafka.producer;
import org.apache.kafka.clients.producer.Partitioner;
import org.apache.kafka.common.Cluster;
import java.util.Map;

/**
 * 1. 实现接口Partitioner
 * 2. 实现3个方法:partition,close,configure
```

```
 * 3. 编写partition方法,返回分区号
 */
public class MyPartitioner implements Partitioner {
    @Override
    public int partition(String topic,
                         Object key, byte[] keyBytes,
                         Object value, byte[] valueBytes, Cluster cluster) {

        String msgValue = value.toString(); // 获取消息
        int partition; // 创建partition
        // 判断消息是否包含atguigu
        if (msgValue.contains("atguigu"))
            partition = 0;
        else
            partition = 1;
        return partition; // 返回分区号
    }
    // 关闭资源
    @Override
    public void close() {}
    // 配置方法
    @Override
    public void configure(Map<String, ?> configs) {}
}
```

- 使用分区器的方法，在生产者的配置中添加分区器参数

```
// 添加自定义分区器
properties.put(ProducerConfig.PARTITIONER_CLASS_CONFIG,"com.atguigu.kafka.producer.MyPartitioner");
```

3.6 生产者提高效率

- 生产者如何提高吞吐量
  - linger.ms：等待时间，修改为5-100ms
  - compression.type：压缩snappy
  - RecordAccumulator：缓冲区大小，修改为64m

```
// batch.size：批次大小，默认16K
properties.put(ProducerConfig.BATCH_SIZE_CONFIG, 16384);

// linger.ms：等待时间，默认0
properties.put(ProducerConfig.LINGER_MS_CONFIG, 1);

// RecordAccumulator：缓冲区大小，默认32M：buffer.memory
properties.put(ProducerConfig.BUFFER_MEMORY_CONFIG, 33554432);

// compression.type：压缩，默认none，可配置值gzip、snappy、lz4和zstd
properties.put(ProducerConfig.COMPRESSION_TYPE_CONFIG,"snappy");
```

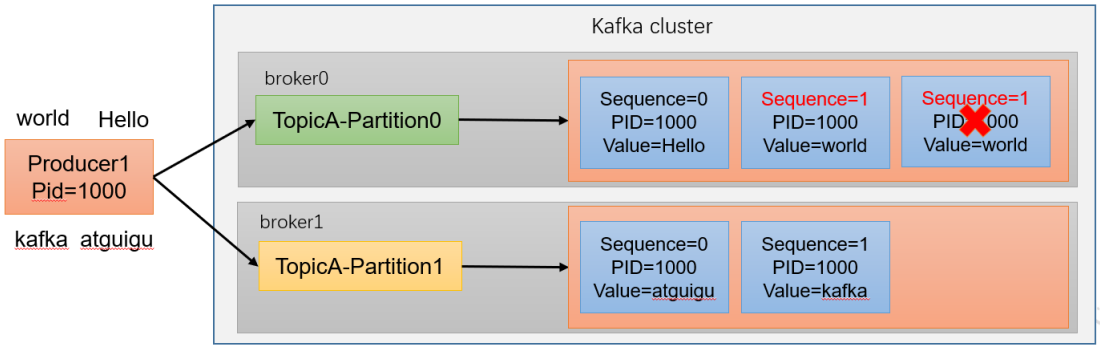
- 数据可靠性
  - acks=0：生产者发送过来的数据，不需要等数据落盘应答
  - acks=1：生产者发送过来的数据，Leader收到数据后应答
  - acks=-1/all：生产者发送过来的数据，Leader和ISR队列里面的所有节点收齐数据后应答
- 如果Leader收到数据，所有Follower都开始同步数据，但有一个Follower，因为某种故障，迟迟不能与Leader进行同步怎么办
  - Leader维护了一个动态的in-sync replica set(ISR), 意为和Leader保持同步的Leader+Follower集合(leader：0，isr:0,1,2)
  - 如果Follower长时间未向Leader发送通信请求或同步数据，则该Follower将被踢出ISR
  - 该时间阈值由replica.lag.time.max.ms参数设定，默认30s
  - 这样就不用等长期联系不上或者已经故障的节点
- 数据可靠性分析：
  - 如果分区副本设置为1个，或者ISR里应答的最小副本数量（ min.insync.replicas 默认为1）设置为1，和ack=1的效果是一样的，仍然有丢数的风险（leader：0，isr:0）
  - 数据完全可靠条件 = ACK级别设置为-1 + 分区副本大于等于2 + ISR里应答的最小副本数量大于等于2
- 可靠性总结：
  - acks=0，生产者发送过来数据就不管了，可靠性差，效率高；
  - acks=1，生产者发送过来数据Leader应答，可靠性中等，效率中等；
  - acks=-1/all，生产者发送过来数据Leader和ISR队列里面所有Follwer应答，可靠性高，效率低；
  - 在生产环境中，acks=0很少使用；acks=1，一般用于传输普通日志，允许丢个别数据；acks=-1，一般用于传输和钱相关的数据，对可靠性要求比较高的场景
- 数据重复分析：
  - acks=-1：生产者发送过来的数据，Leader和ISR队列里面的所有节点收齐数据后应答。
  - 如果此时leader挂了，follower成为leader，那么需要重复接收一份数据
  - 代码配置

```
// 设置acks： 0, 1, all/-1
properties.put(ProducerConfig.ACKS_CONFIG, "all");

// 重试次数retries，默认是int最大值，2147483647
properties.put(ProducerConfig.RETRIES_CONFIG, 3);
```

3.7 生产者幂等性

- 数据传递语义
  - 至少一次（At Least Once）= ACK级别设置为 -1 + 分区副本>=2 + ISR里应答的最小副本数量>=2
  - 最多一次（At Most Once）= ACK级别设置为 0
  - 总结：

        - At Least Once可以保证数据不丢失，但是不能保证数据不重复
        - At Most Once可以保证数据不重复，但是不能保证数据不丢失
    - 精确一次（Exactly Once）：对于一些非常重要的信息，比如和钱相关的数据，要求数据既不能重复也不丢失
- 幂等性 idempotence
    - 幂等性就是指Producer不论向Broker发送多少次重复数据，Broker端都只会持久化一条，保证不重复
    - 精确一次（Exactly Once） = 幂等性 + 至少一次（ ack=-1 + 分区副本数>=2 + ISR最小副本数量>=2）
    - 重复数据的判断标准：具有**<PID, Partition, SeqNumber>**相同主键的消息提交时，Broker只会持久化一条
    - 其中PID是Kafka每次重启都会分配一个新的；Partition 表示分区号；Sequence Number是单调自增的。
    - 所以幂等性只能保证的是在单分区单会话内不重复

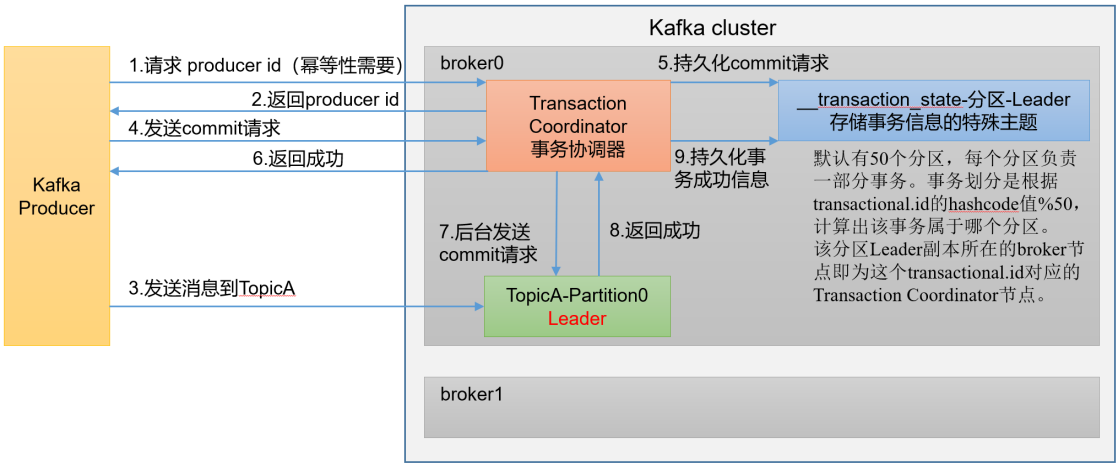


    - 开启参数enable.idempotence，设置成true后，Producer自动升级成幂等性Producer

```
props.put(“enable.idempotence”, ture) //或者
props.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, true)
```

3.8 生产者事务

- Kafka提供了对事务的支持，能保证多条消息原子性地写入到目标分区，同时也保证Consumer只能看到事务成功提交的消息
    - 要么全部写入成功，要么全部失败
    - 没有事务，重启kafka会有重复数据。事务型Producer不惧进程的重启
- 设置事务，两个要求
    - 开启参数enable.idempotence = true
    - 设置Producer端参数transactional.id
- Producer 在使用事务功能前，必须先自定义一个唯一的 **transactional.id**
    - 有了 transactional.id，即使客户端挂掉了，它重启后也能继续处理未完成的事务



- Kafka的事务5个API

```
// 1初始化事务
void initTransactions();
// 2开启事务
void beginTransaction() throws ProducerFencedException;
// 3在事务内提交已经消费的偏移量（主要用于消费者）
void sendOffsetsToTransaction(Map<TopicPartition, OffsetAndMetadata> offsets,
                              String consumerGroupId) throws
ProducerFencedException;
// 4提交事务
void commitTransaction() throws ProducerFencedException;
// 5放弃事务（类似于回滚事务的操作）
void abortTransaction() throws ProducerFencedException;
```

- 单个Producer，使用事务保证消息的仅一次发送

```
// 设置事务id（必须），事务id任意起名
properties.put(ProducerConfig.TRANSACTIONAL_ID_CONFIG, "transaction_id_0");
// 3. 创建kafka生产者对象
KafkaProducer<String, String> kafkaProducer = new KafkaProducer<String, String>
(properties);
// 初始化事务
kafkaProducer.initTransactions();
// 开启事务
kafkaProducer.beginTransaction();
try {
    // 4. 调用send方法,发送消息
    for (int i = 0; i < 5; i++) {
        // 发送消息
        kafkaProducer.send(new ProducerRecord<>("first", "atguigu " + i));
    }
    // 提交事务
    kafkaProducer.commitTransaction();
} catch (Exception e) {
    // 终止事务
    kafkaProducer.abortTransaction();
} finally {
```

```
    // 5. 关闭资源
    kafkaProducer.close();
}
```

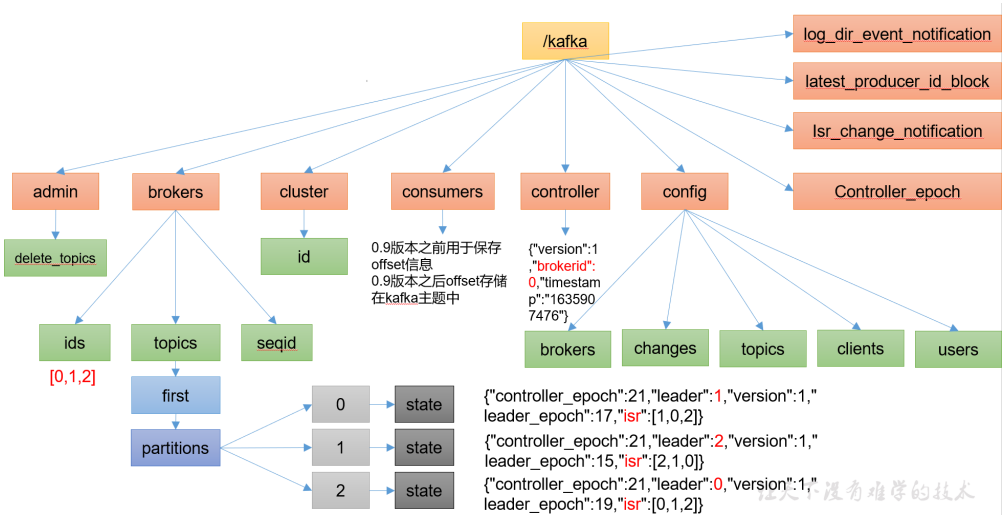
3.8 数据有序和乱序

- 数据有序
  - 单分区内，有序；多分区，分区与分区间无序
  - 多分区有序实现需要consumer端统一排序，但是效率低要等待所有数据
- 数据乱序
  - kafka在1.x版本之前保证数据单分区有序：max.in.flight.requests.per.connection=1
  - kafka在1.x及以后版本保证数据单分区有序
    - 未开启幂等性 max.in.flight.requests.per.connection需要设置为1
    - 开启幂等性 max.in.flight.requests.per.connection需要设置小于等于5
      - 原因说明：因为在kafka1.x以后，启用幂等性后，kafka服务端会缓存producer发来的最近5个request的元数据(有Sequence Number)，故无论如何，都可以保证最近5个request的数据都是有序的
      - 如果开启了幂等性且缓存的请求个数小于5个。会在服务端重新排序

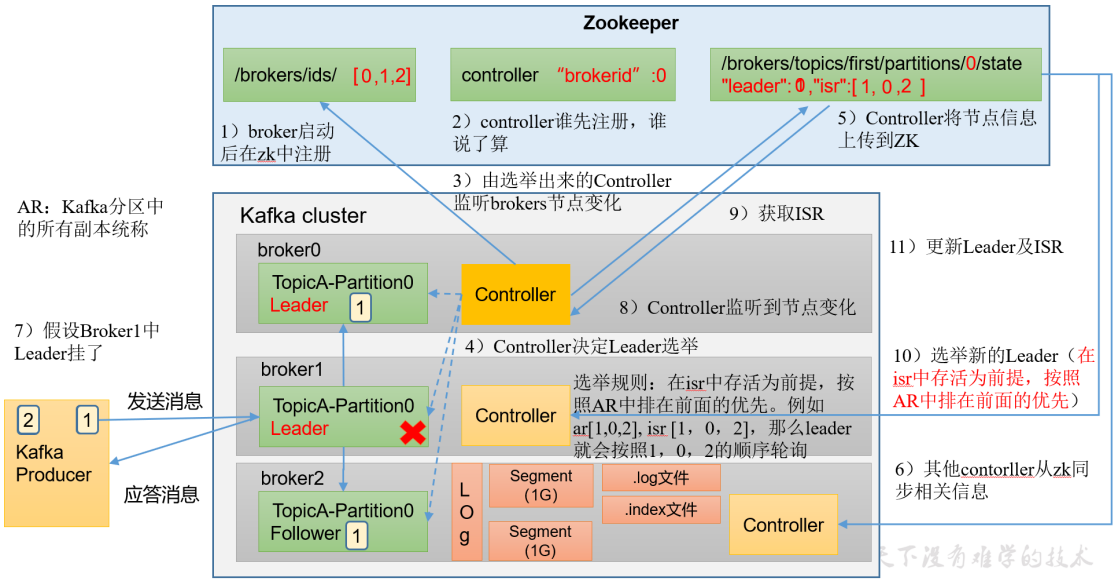
4. Kafka Broker

4.1 Broker 总体工作流程

- Zookeeper存储的Kafka信息
  - 启动Zookeeper客户端 `bin/zkCli.sh`
  - 通过ls命令可以查看kafka相关信息 `ls /kafka`
  - 可视化工具：PRETTYZOO
    - 信息：
      - /kafka/brokers/ids [0,1,2] 记录有哪些服务器
      - /kafka/brokers/topics/first/partitions/0/state {"leader":1 ,"isr":[1,0,2] } 记录谁是Leader，有哪些服务器可用
      - /kafka/controller {“brokerid”:0} 辅助选举Leader



- 总体工作流程
  - broker启动后在zookeeper中注册
  - 每个broker有一个controller，谁先注册，谁成为leader
  - 由选举出来的Controller监听brokers节点变化
  - Controller决定Leader选举
  - Controller将节点信息上传到ZK
  - 其他contorller从zk同步相关信息
  - 假设Broker1中Leader挂了 - Controller监听到节点变化 - 获取ISR - 选举新的Leader



4.2 生产经验 - 服役新节点

- 新节点准备 (克隆104)
  - 关闭hadoop104，并右键执行克隆操作
  - 开启hadoop105，并修改IP地址
  - 在hadoop105上，修改主机名称为hadoop105
  - 重新启动hadoop104、hadoop105
  - 修改haodoop105中kafka的broker.id为3

- 删除hadoop105中kafka下的datas和logs
- 启动hadoop102、hadoop103、hadoop104上的kafka集群
- 单独启动hadoop105中的kafka

- 执行负载均衡操作
  - 执行负载均衡操作

```
vim topics-to-move.json
{
    "topics": [
        {"topic": "first"}
    ],
    "version": 1
}
```

  - 生成一个负载均衡的计划

```
// 对first主题进行负载均衡
bin/kafka-reassign-partitions.sh --bootstrap-server hadoop102:9092  --
topics-to-move-json-file topics-to-move.json --broker-list "0,1,2,3" --
generate
```

  - 创建副本存储计划（所有副本存储在broker0、broker1、broker2、broker3中）

```
vim increase-replication-factor.json
//输入刚刚生成的计划
{"version":1,"partitions":[{"topic":"first","partition":0,"replicas":
[2,3,0],"log_dirs":["any","any","any"]},
{"topic":"first","partition":1,"replicas":[3,0,1],"log_dirs":
["any","any","any"]},{"topic":"first","partition":2,"replicas":
[0,1,2],"log_dirs":["any","any","any"]}]}
```

  - 执行副本存储计划

```
bin/kafka-reassign-partitions.sh --bootstrap-server hadoop102:9092 --
reassignment-json-file increase-replication-factor.json --execute
```

  - 验证副本存储计划

```
bin/kafka-reassign-partitions.sh --bootstrap-server hadoop102:9092 --
reassignment-json-file increase-replication-factor.json --verify
```

4.3 生产经验 - 退役旧节点

- 执行负载均衡操作
  - 先按照退役一台节点，生成执行计划，然后按照服役时操作流程执行负载均衡
  - 创建一个要均衡的主题
  - 创建执行计划

    ```
    bin/kafka-reassign-partitions.sh --bootstrap-server hadoop102:9092  --
    topics-to-move-json-file topics-to-move.json --broker-list "0,1,2" --
    generate
    ```

  - 创建副本存储计划（所有副本存储在broker0、broker1、broker2中）
  - 执行副本存储计划
  - 验证副本存储计划
- 执行停止命令 (停止105)
  - 在hadoop105上执行停止命令即可`bin/kafka-server-stop.sh`

4.4 副本机制

- 同一个分区下的所有副本保存有相同的消息序列，这些副本分散保存在不同的Broker上，从而对抗Broker宕机带来的数据不可用
- Kafka中，副本分成两类：Leader Replica 和 Follower Replica
- 每个分区在创建时都要选举一个副本，称为Leader Replica，其余的副本自动称为Follower Replica
- Leader 基本概念
  - Kafka生产者只会把数据发往Leader, 然后Follower 找Leader进行同步数据
  - 所有的读写请求都必须发往Leader 所在的Broker，由该Broker负责处理
- Follower 基本概念
  - Kafka副本作用：**提高数据可靠性**
  - Kafka默认 Follower 副本1个, 生产环境一般配置为2个, 保证数据可靠性
  - 太多副本会增加磁盘存储空间, 增加网络上数据传输, 降低效率
  - Follower 不对外提供服务的，任何一个Follower 都不能响应消费者和生产者的读写请求
  - Follower 唯一的任务就是从Leader **「异步拉取」**消息，并写入到自己的提交日志中，从而实现与Leader 的同步
- 副本机制两个好处
  - 方便实现Read-your-writes
    - 当使用生产者API向Kafka成功写入消息后，马上使用消费者API去读取刚才生产的消息
  - 方便实现单调读 Monotonic Reads

        - 假设当前有2个追随者副本F1和F2，它们异步地拉取领导者副本数据。倘若F1拉取了Leader的最新消息而F2还未及时拉取，如果有消费者先从F1读取消息之后又从F2拉取消息，它可能会看到这样的现象：第一次消费时看到的最新消息在第二次消费时不见了，这就不是单调读一致性
        - 所有的读请求都是由Leader来处理，那么Kafka就很容易实现单调读一致性

4.5 ISR 机制

- Kafka分区中的所有副本统称为AR (Assigned Repllicas)
    - AR = ISR + OSR
    - ISR (In-sync Replicas) 表示和Leader保持同步的Follower集合。如果Follower长时间未向Leader发送通信请求或同步数据，则该Follower将被踢出ISR。该时间阈值由replica.lag.time.max.ms参数设定，默认30s。如果Leader发生故障，就会从ISR中选举新的Leader
    - ISR不只是Follower 集合，它必然包括Leader副本。甚至在某些情况下，ISR只有Leader这一个副本
    - OSR 表示Follower与Leader副本不同步的，延迟过多的副本
- Leader选举流程
    - 如果一个分区的leader副本不可用，就意味着整个分区不可用，此时需要从follower副本中选举出新的leader副本提供服务
    - Kafka集群中有一个broker的Controller会被选举为Controller Leader，负责管理集群broker的上下线，所有topic的分区副本分配和Leader选举等工作
    - 而对于Leader选举，如果非同步Follower 落后Leader太多，被选为新的Leader，就可能出现数据的丢失
    - 选举规则：
        - 会尽量分配每个partition副本leader在不同的broker中，避免多个leader在同一个broker，导致集群中的broker负载不平衡
        - 在ISR 中存活为前提，按照AR中排在前面的优先。例如 AR [1,0,2], ISR [1,0,2]，那么leader就会按照1,0,2的顺序轮询
    - Controller的信息同步工作是依赖于Zookeeper的
- Leader Partition负载平衡
    - 正常情况下，Kafka本身会自动把Leader Partition均匀分散在各个机器上，来保证每台机器的读写吞吐量都是均匀的
    - 但是如果某些broker宕机，会导致Leader Partition过于集中在其他少部分几台broker上，这会导致少数几台broker的读写请求压力过高，其他宕机的broker重启之后都是follower partition，读写请求很低，造成集群负载不均衡
    - 解决方法
        - auto.leader.rebalance.enable，默认是true。自动Leader Partition 平衡
        - leader.imbalance.per.broker.percentage，默认是10%。每个broker允许的不平衡的leader的比率。如果每个broker超过了这个值，控制器会触发leader的平衡
        - leader.imbalance.check.interval.seconds，默认值300秒。检查leader负载是否平衡的间隔时间

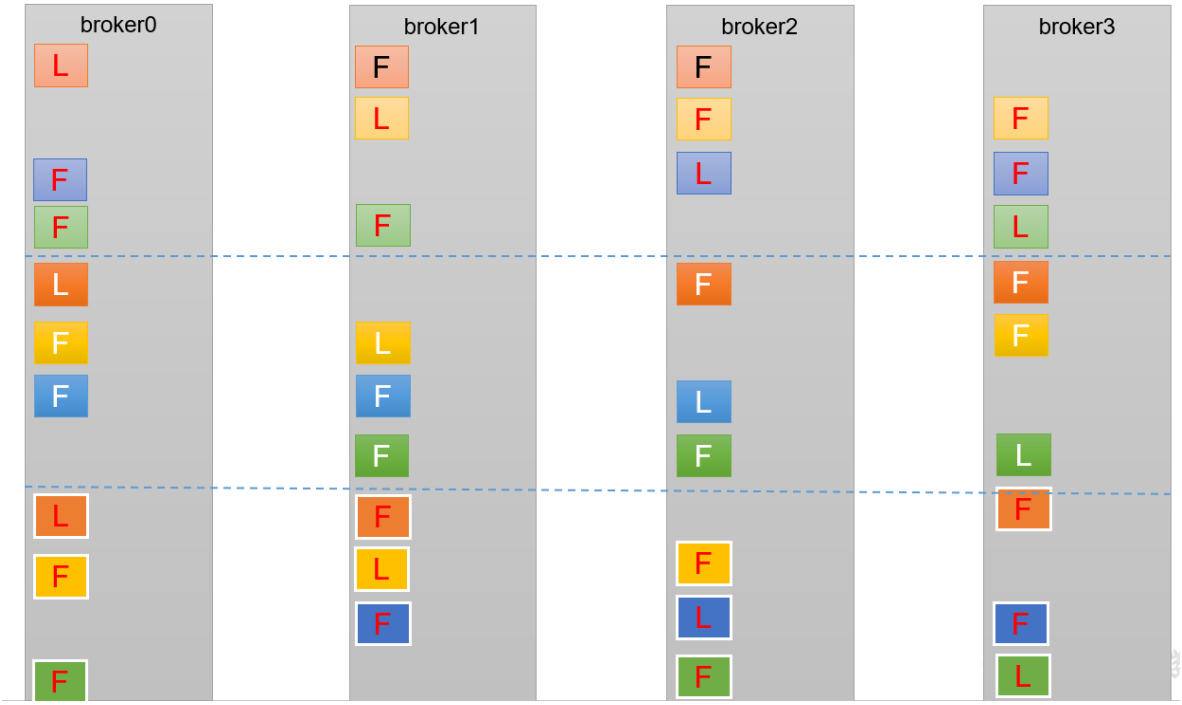
    - 计算不平衡的leader的比率
        - 针对broker0节点，分区2的AR优先副本是0节点，但是0节点却不是Leader节点，所以不平衡数加1
        - AR副本总数是4，broker0节点不平衡率为1/4>10%，需要再平衡

4.6 Leader 和 Follower故障

- LEO（Log End Offset）：每个副本的最后一个offset，LEO其实就是最新的offset + 1。
- HW（High Watermark）：所有副本中最小的LEO 。
- Follower故障处理细节
    - Follower发生故障后会被临时踢出ISR
    - 这个期间Leader和Follower继续接收数据
    - 待该Follower恢复后，Follower会读取本地磁盘记录的上次的HW，并将log文件高于HW的部分截取掉，从HW开始向Leader进行同步
    - 等该Follower的LEO大于等于该Partition的HW，即Follower追上Leader之后，就可以重新加入ISR了
- Leader故障处理细节
    - Leader发生故障之后，会从ISR中选出一个新的Leader
    - 为保证多个副本之间的数据一致性，其余的Follower会先将各自的log文件高于HW的部分截掉，然后从新的Leader同步数据
- 注意：这只能保证副本之间的数据一致性，并不能保证数据不丢失或者不重复

4.7 分区副本分配

- 如果kafka服务器只有4个节点但设置了16个分区，那么设置kafka的分区数大于服务器台数，在kafka底层如何分配存储副本
- `bin/kafka-topics.sh --bootstrap-server hadoop102:9092 --describe --topic second`
- 尽可能保证负载均衡，轮询



4.8 手动调整分区Follower存储

- 在生产环境中，每台服务器的配置和性能不一致，但是Kafka只会根据自己的代码规则创建对应的分区副本，就会导致个别服务器存储压力较大。所有需要**手动调整**分区副本的存储
- 需求：创建一个新的topic，4个分区，两个副本，名称为three。将该topic的所有副本都存储到broker0和broker1两台服务器上
  - 创建一个新的topic，名称为three

    ```
    bin/kafka-topics.sh --bootstrap-server hadoop102:9092 --create --partitions
    4 --replication-factor 2 --topic three
    ```

  - 查看分区副本存储情况

    ```
    bin/kafka-topics.sh --bootstrap-server hadoop102:9092 --describe --topic
    three
    ```

  - 创建副本存储计划（所有副本都指定存储在broker0、broker1中）

    ```
    vim increase-replication-factor.json
    {
        "version":1,
        "partitions":[{"topic":"three","partition":0,"replicas":[0,1]},
                    {"topic":"three","partition":1,"replicas":[0,1]},
                    {"topic":"three","partition":2,"replicas":[1,0]},
                    {"topic":"three","partition":3,"replicas":[1,0]}]
    }
    ```

  - 执行副本存储计划
  - 验证副本存储计划

- 查看分区副本存储情况

```
bin/kafka-topics.sh --bootstrap-server hadoop102:9092 --describe --topic
three
```

4.9 生产经验 - 增加 Follower 因子

- 在生产环境当中，由于某个主题的重要等级需要提升，我们考虑增加副本。
- 副本数的增加需要先制定计划，然后根据计划执行
  - 创建topic
  - 手动增加副本存储
    - 创建副本存储计划（所有副本都指定存储在broker0、broker1、broker2中）
    - 执行副本存储计划

4.10 文件存储机制

- Topic数据的存储机制
  - Topic是逻辑上的概念, 而partition是物理上的概念
  - 每个partition对应于一个log文件, log文件中存储的就是Producer生产的数据
  - Producer生产的数据会被不断追加到该log文件末端, 为防止log文件过大导致数据定位效率低下, Kafka采取了分片和索引机制, 将每个partition分为多个segment
  - 每个segment包括：“.index”文件、“.log”文件和.timeindex等文件
    - 这些文件位于一个文件夹下，该文件夹的命名规则为：topic名称+分区序号，例如：first-0
- Topic数据到底存储在什么位置
  - /opt/module/kafka/datas/first-1
- index文件和log文件详解
  - index为稀疏索引，大约每往log文件写入4kb数据，会往index文件写入一条索引。参数log.index.interval.bytes默认4kb
  - Index文件中保存的offset为相对offset，这样能确保offset的值所占空间不会过大，因此能将offset的值控制在固定大小
- 如何在log文件中定位到offset=600的Record
  - 根据目标offset定位Segment文件
  - 找到小于等于目标offset的最大offset对应的索引项
  - 定位到log文件
  - 向下遍历找到目标Record

4.11 文件清理策略

- Kafka中**默认的日志保存时间为7天**，可以通过调整如下参数修改保存时间
  - log.retention.hours，最低优先级小时，默认7天。
  - log.retention.minutes，分钟。
  - log.retention.ms，最高优先级毫秒。
  - log.retention.check.interval.ms，负责设置检查周期，默认5分钟
- 那么日志一旦超过了设置的时间，怎么处理
  - Kafka中提供的日志清理策略有delete和compact两种
- delete日志删除：将过期数据删除
  - log.cleanup.policy = delete 所有数据启用删除策略
  - 基于时间：默认打开。以segment中所有记录中的最大时间戳作为该文件时间戳
  - 基于大小：默认关闭。超过设置的所有日志总大小，删除最早的segment

    log.retention.bytes，默认等于-1，表示无穷大
- compact日志压缩
  - compact日志压缩：对于相同key的不同value值，只保留最后一个版本
  - log.cleanup.policy = compact 所有数据启用压缩策略
  - 压缩后的offset可能是不连续的，比如上图中没有6，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，实际上会拿到offset为7的消息，并从这个位置开始消费
  - 这种策略只适合特殊场景，比如消息的key是用户ID，value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

4.12 高效读写数据

- Kafka本身是分布式集群，可以采用分区技术，并行度高
- 读数据采用稀疏索引，可以快速定位要消费的数据
- 顺序写磁盘
  - Kafka的producer生产数据，要写入到log文件中，写的过程是一直追加到文件末端，为顺序写
  - 官网有数据表明，同样的磁盘，顺序写能到600M/s，而随机写只有100K/s
  - 这与磁盘的机械机构有关，顺序写之所以快，是因为其省去了大量磁头寻址的时间
- 页缓存 + 零拷贝技术
  - **零拷贝**：Kafka的数据加工处理操作交由Kafka生产者和Kafka消费者处理。Kafka Broker应用层不关心存储的数据，所以就不用走应用层，传输效率高
  - **PageCache页缓存**：Kafka重度依赖底层操作系统提供的PageCache功能。当上层有写操作时，操作系统只是将数据写入PageCache。当读操作发生时，先从PageCache中查找，如果找不到，再去磁盘中读取。实际上PageCache是把尽可能多的空闲内存都当做了磁盘缓存来使用

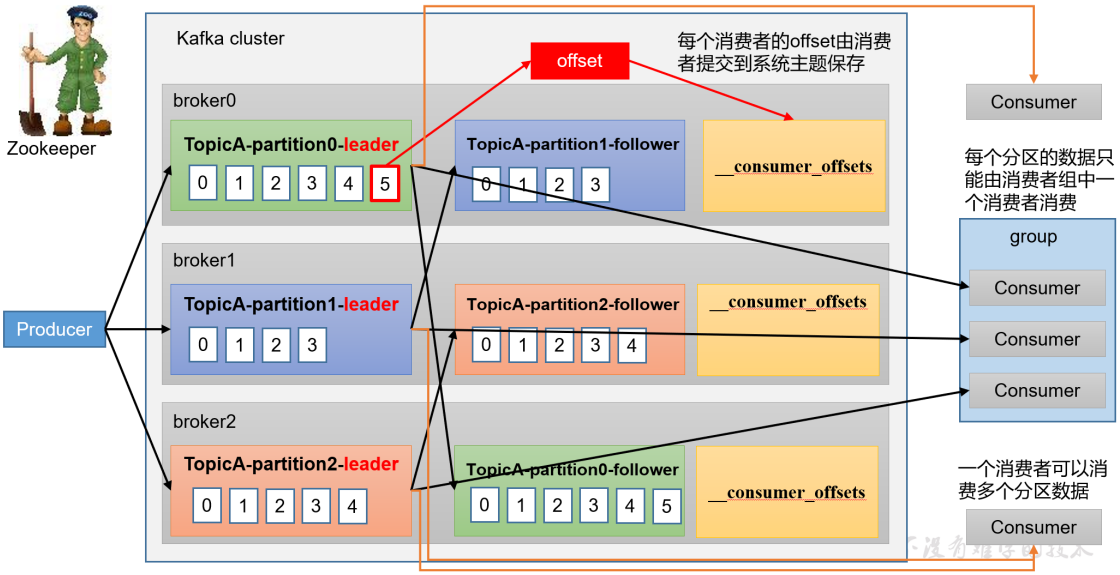
5. Kafka 消费者

5.1 Kafka消费方式

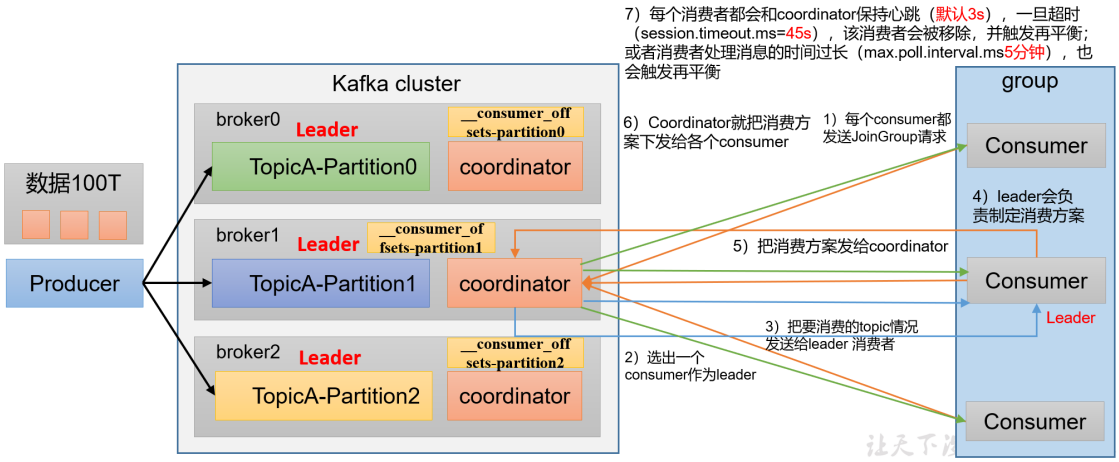
- pull模式(拉取)
  - 每个consumer拉取速度不一样，kafka采用从broker中主动拉取数据
  - 不足之处是，如果Kafka没有数据，消费者可能会陷入循环中，一直返回空数据
- push模式(推送)
  - Kafka没有采用这种方式，因为由broker决定消息发送速率，很难适应所有消费者的消费速率
  - 拉取速度慢的consumer来不及处理消息

5.2 Kafka消费者工作流程

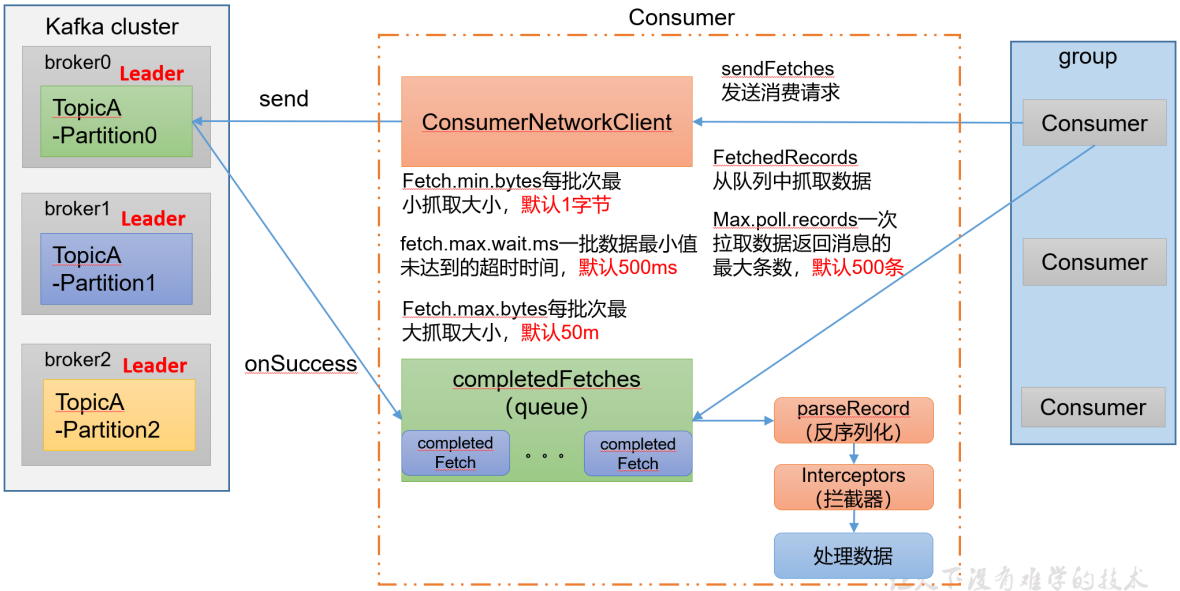
- 总体工作流程
  - producer向每个partition的leader发送数据，follower主动跟leader同步数据
  - consumer可以消费一个或多个partition的数据，但是消费者组只能当成一个消费者
  - consumer消费进度(offset)维护在broker



- 消费者组原理
  - Consumer Group(CG): 消费者组, 由多个consumer组成。形成一个消费者组的条件，是所有消费者的groupid相同
  - 消费者组内每个消费者负责消费不同分区的数据，一个分区只能由一个组内消费者消费
  - 消费者组之间互不影响。所有的消费者都属于某个消费者组，即消费者组是逻辑上的一个订阅者
  - 如果向消费组中添加更多的消费者，超过主题分区数量，则有一部分消费者就会闲置，不会接收任何消息
- 消费者组初始化流程
  - coordinator：辅助实现消费者组的初始化和分区的分配
    - coordinator节点选择 = groupid的hashcode值 % 50（__consumer_offsets的分区数量）
    - 例如： groupid的hashcode值 = 1，1% 50 = 1，那么__consumer_offsets 主题的1号分区，在哪个broker上，就选择这个节点的coordinator作为这个消费者组的老大。消费者组下的所有的消费者提交offset的时候就往这个分区去提交offset



- 消费流程
  - consumer 先创建 ConsumerNetworkClient



5.3 消费者API

- 独立消费者订阅主题
  - 创建一个独立消费者，消费first主题中数据
  - 注意：在消费者API代码中必须配置消费者组id。命令行启动消费者不填写消费者组id会被自动填写随机的消费者组id

```
package com.atguigu.kafka.consumer;

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

import java.time.Duration;
import java.util.ArrayList;
import java.util.Properties;
```

```
public class CustomConsumerTopic {
    public static void main(String[] args) {
        // 1.创建消费者的配置对象
        Properties properties = new Properties();
        // 2.给消费者配置对象添加参数
        properties.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
"hadoop102:9092");
        // 配置序列化 必须
        properties.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
        properties.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
        // 配置消费者组(组名任意起名) 必须
        properties.put(ConsumerConfig.GROUP_ID_CONFIG, "test");
        // 创建消费者对象
        KafkaConsumer<String, String> kafkaConsumer = new
KafkaConsumer<String, String>(properties);
        // 注册要消费的主题(可以消费多个主题)
        ArrayList<String> topics = new ArrayList<>();
        topics.add("first");
        // 订阅主题
        kafkaConsumer.subscribe(topics);
        // 拉取数据打印
        while (true) {
            // 设置1s中消费一批数据
            ConsumerRecords<String, String> consumerRecords =
                kafkaConsumer.poll(Duration.ofSeconds(1));
            // 打印消费到的数据
            for (ConsumerRecord<String, String> consumerRecord :
consumerRecords) {
                System.out.println(consumerRecord);
            }
        }
    }
}
```

- 独立消费者订阅分区
  - 创建一个独立消费者，消费first主题0号分区的数据

```
public class CustomConsumerPartition {
    public static void main(String[] args) {
        // 1.创建消费者的配置对象
        Properties properties = new Properties();
        // 2.给消费者配置对象添加参数

 properties.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,"hadoop102:9092");
        // 配置序列化 必须
        properties.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
        properties.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
```

```
        // 配置消费者组（必须），名字可以任意起
        properties.put(ConsumerConfig.GROUP_ID_CONFIG,"test");
        // 创建消费者对象
        KafkaConsumer<String, String> kafkaConsumer = new KafkaConsumer<>
(properties);
        // 消费某个主题的某个分区数据
        ArrayList<TopicPartition> topicPartitions = new ArrayList<>();
        topicPartitions.add(new TopicPartition("first", 0));
        // 订阅分区
        kafkaConsumer.assign(topicPartitions);
        while (true){
            ConsumerRecords<String, String> consumerRecords =
                kafkaConsumer.poll(Duration.ofSeconds(1));
            for (ConsumerRecord<String, String> consumerRecord :
consumerRecords) {
                System.out.println(consumerRecord);
            }
        }
    }
}
```

- 消费者组
  - 测试同一个主题的分区数据，只能由一个消费者组中的一个消费
  - 因为GROUP_ID设置的一样，所以在run时自动成为一个组

5.4 Partition分配

- 一个 Consumer Group 中有多个Consumer 组成，一个 topic有多个partition组成
  - Group ID是一个字符串，在一个Kafka集群中，它标识唯一的一个Consumer Group
  - Consumer Group之间彼此独立，互不影响，它们能够订阅相同的一组主题而互不干涉
  - 到底由哪个Consumer 来消费哪个partition的数据?
- Kafka有四种主流的分区分配策略： **Range、RoundRobin、Sticky、CooperativeSticky**
  - 可以通过配置参数partition.assignment.strategy，修改分区的分配策略
  - 默认策略是 Range + CooperativeSticky
  - Kafka可以同时使用多个分区分配策略
  - 分配策略就是初始化consumer时候，会向coordinator汇报，coordinator决定一个leader，leader负责制定消费方案
- Range 分配策略
  - 对一个消费者组来说决定消费方式是以分区总数除以消费者总数来决定
  - 首先对同一个 topic 里面的分区按照序号进行排序，并对消费者按照字母顺序进行排序
  - 假如现在有 7 个分区，3 个消费者，排序后的分区将会是0,1,2,3,4,5,6；消费者排序完之后将会是C0,C1,C2
  - 通过 partitions/consumer数 来决定每个消费者应该消费几个分区。如果除不尽，那么前面几个消费者将会多消费 1 个分区

    - 例如，7/3 = 2 余 1 ，除不尽，那么 消费者 C0 便会多消费 1 个分区。 8/3=2余2，除不尽，那么C0和C1分别多消费一个
    - 注意：如果只是针对 1 个 topic 而言，C0消费者多消费1个分区影响不是很大。但是如果有 N 多个 topic，那么针对每个 topic，消费者 C0都将多消费 1 个分区，topic越多，C0消费的分区会比其他消费者明显多消费 N 个分区。
    - 容易产生数据倾斜
- RoundRobin 分配策略
    - RoundRobin 轮循说的是对于同一组消费者来说，使用轮训分配的方式，决定消费者消费的分区
    - 把所有的 partition 和所有的 consumer 都列出来，按照 hashcode 进行排序，通过轮询算法来分配 partition 给各个消费者
- Sticky 分配策略
    - 在同组中有新的消费者加入或者旧的消费者退出时，不会从新开始Range分配，而是保留现有消费策略，将退出的消费者所消费的分区平均分配给现有消费者；新增消费者则从其他现存消费者的消费策略中分离
    - Sticky 粘性分区定义：可以理解为分配的结果带有“粘性的”
        - 即在执行一次新的分配之前，考虑上一次分配的结果，尽量少的调整分配的变动，可以节省大量的开销
    - 粘性分区是Kafka从0.11.x版本开始引入这种分配策略，首先会尽量均衡的放置分区到消费者上面，在出现同一消费者组内消费者出现问题的时候，会尽量保持原有分配的分区不变化

```
// 修改分区分配策略
ArrayList<String> startegys = new ArrayList<>();
startegys.add("org.apache.kafka.clients.consumer.StickyAssignor");

properties.put(ConsumerConfig.PARTITION_ASSIGNMENT_STRATEGY_CONFIG,
startegys);
```

5.5 Offset 位移

- 默认维护位置
    - producer向每个 partition的leader发送数据，consumer消费一个或多个partition的数据
    - 消费的过程中会有一个消费的offset，表示消费的位置进度
    - Kafka0.9版本之前，consumer默认将offset保存在Zookeeper中；从0.9版本开始，consumer默认将offset保存在Kafka一个内置的topic中，该topic为`__consumer_offsets`
    - 当Kafka集群中的第一个Consumer程序启动时，Kafka会自动创建位移主题, 默认该主题的分区数是50，副本数是3
    - `__consumer_offsets`主题里面采用key和value的方式存储数据。key是group.id+topic+分区号，value就是当前offset的值。每隔一段时间，kafka内部会对这个topic进行compact，也就是每个group.id+topic+分区号就保留最新数据
    - 因为Consumer能够同时消费多个分区的数据，Consumer需要为分配给它的每个分区提交各自的位移数据
- 自动提交offset

- 为了使我们能够专注于自己的业务逻辑，Kafka提供了自动提交offset的功能
- 自动提交offset的相关参数：
  - enable.auto.commit：是否开启自动提交offset功能，默认是true
  - auto.commit.interval.ms：自动提交offset的时间间隔，默认是5s
- 不断的拉取数据，Consumer每5s自动提交一次offset
- 配置自动提交offset

```
// 配置消费者组
properties.put(ConsumerConfig.GROUP_ID_CONFIG, "test");
// 是否自动提交offset
properties.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, true);
// 提交offset的时间周期1000ms，默认5s
properties.put(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG, 1000);
```

- 手动提交offset
  - 虽然自动提交offset十分简单便利，但由于其是基于时间提交的，开发人员难以把握offset提交的时机
    - **因此Kafka还提供了手动提交offset的API**
  - 手动提交offset的方法有两种：分别是**commitSync(同步提交)**和**commitAsync(异步提交)**
  - 两者的相同点是，都会将本次提交的一批数据最高的偏移量提交；不同点是，同步提交阻塞当前线程，一直到提交成功，并且会自动失败重试（由不可控因素导致，也会出现提交失败）；而异步提交则没有失败重试机制，故有可能提交失败
    - commitSync(同步提交): 必须等待offset提交完毕，再去消费下一批数据

```
// 配置消费者组
properties.put(ConsumerConfig.GROUP_ID_CONFIG, "test");
// 是否自动提交offset
properties.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);
// ....
// 5. 消费数据
while (true){
    // 读取消息
    ConsumerRecords<String, String> consumerRecords =
consumer.poll(Duration.ofSeconds(1));
    // 输出消息
    for (ConsumerRecord<String, String> consumerRecord :
consumerRecords) {
        System.out.println(consumerRecord.value());
    }
    // 同步提交offset
    consumer.commitSync();
}
```

    - commitAsync(异步提交): 发送完提交offset请求后，就开始消费下一批数据了

```
// 配置消费者组
```

```
properties.put(ConsumerConfig.GROUP_ID_CONFIG, "test");
// 是否自动提交offset
properties.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);
// ....
// 5. 消费数据
while (true){
    // 读取消息
    ConsumerRecords<String, String> consumerRecords =
consumer.poll(Duration.ofSeconds(1));
    // 输出消息
    for (ConsumerRecord<String, String> consumerRecord :
consumerRecords) {
        System.out.println(consumerRecord.value());
    }
    // 异步提交offset
    consumer.commitAsync();
}
```

- 指定Offset消费
  - 参数: auto.offset.reset = earliest | latest | none 默认是latest
  - 当Kafka中没有初始偏移量 (消费者组第一次消费) 或服务器上不再存在当前偏移量时 (例如该数据已被删除) 该怎么办
    - earliest：自动将偏移量重置为最早的偏移量，--from-beginning
    - latest（默认值）：自动将偏移量重置为最新偏移量
    - none：如果未找到消费者组的先前偏移量，则向消费者抛出异常

```
// 0  配置信息
Properties properties = new Properties();
// ...
// 1 创建一个消费者
KafkaConsumer<String, String> kafkaConsumer = new KafkaConsumer<>
(properties);
// 2 订阅一个主题
ArrayList<String> topics = new ArrayList<>();
topics.add("first");
kafkaConsumer.subscribe(topics);

// 指定位置消费
Set<TopicPartition> assignment = new HashSet<>();
while (assignment.size() == 0) {
    kafkaConsumer.poll(Duration.ofSeconds(1));
    // 获取消费者分区分配信息（有了分区分配信息才能开始消费）
    assignment = kafkaConsumer.assignment();
}
    // 遍历所有分区，并指定offset从1700的位置开始消费
for (TopicPartition tp: assignment) {
    kafkaConsumer.seek(tp, 1700);
}
```

```
// 3 消费该主题数据
while (true) {
    ConsumerRecords<String, String> consumerRecords =
kafkaConsumer.poll(Duration.ofSeconds(1));
    for (ConsumerRecord<String, String> consumerRecord : consumerRecords) {
        System.out.println(consumerRecord);
    }
}
```

- 注意：每次执行完，需要修改消费者组名

- 指定时间消费
  - 在生产环境中，会遇到最近消费的几个小时数据异常，想重新按照时间消费
  - 例如要求按照时间消费前一天的数据，怎么处理

```
// 0  配置信息
Properties properties = new Properties();
// ...
// 1 创建一个消费者
KafkaConsumer<String, String> kafkaConsumer = new KafkaConsumer<>
(properties);
// 2 订阅一个主题
ArrayList<String> topics = new ArrayList<>();
topics.add("first");
kafkaConsumer.subscribe(topics);

// 指定时间消费
Set<TopicPartition> assignment = new HashSet<>();
while (assignment.size() == 0) {
    kafkaConsumer.poll(Duration.ofSeconds(1));
    // 获取消费者分区分配信息（有了分区分配信息才能开始消费）
    assignment = kafkaConsumer.assignment();
}
// 把时间转换为对应offset
HashMap<TopicPartition, Long> timestampToSearch = new HashMap<>();
// 封装集合存储，每个分区对应一天前的数据
for (TopicPartition topicPartition : assignment) {
    timestampToSearch.put(topicPartition, System.currentTimeMillis() - 1 *
24 * 3600 * 1000);
}
// 获取从1天前开始消费的每个分区的offset
Map<TopicPartition, OffsetAndTimestamp> offsets =
kafkaConsumer.offsetsForTimes(timestampToSearch);
// 遍历每个分区，对每个分区设置消费时间。
for (TopicPartition topicPartition : assignment) {
    OffsetAndTimestamp offsetAndTimestamp = offsets.get(topicPartition);
    // 根据时间指定开始消费的位置
    if (offsetAndTimestamp != null){
        kafkaConsumer.seek(topicPartition, offsetAndTimestamp.offset());
    }
}
```

```
// 3 消费该主题数据
while (true) {

    ConsumerRecords<String, String> consumerRecords =
kafkaConsumer.poll(Duration.ofSeconds(1));

    for (ConsumerRecord<String, String> consumerRecord : consumerRecords) {
        System.out.println(consumerRecord);
    }
}
```

- 漏消费和重复消费
  - 重复消费：已经消费了数据，但是offset没提交
    - 自动提交offset引起
    - Consumer每5s提交offset，如果提交offset后的2s，consumer挂了，再次重启consumer，则从上一次提交的offset处继续消费，导致重复消费
  - 漏消费：先提交offset后消费，有可能会造成数据的漏消费
    - 手动提交offset引起
    - 当offset被提交时，数据还在内存中未落盘，此时刚好消费者线程被kill掉
    - 那么offset已经提交，但是数据未处理，导致这部分内存中的数据丢失
  - 既不漏消费也不重复消费 => 消费者事务
- Compact策略
  - Kafka使用Compact策略来删除__consumer_offsets主题中的过期消息，避免该主题无限期膨胀
  - Kafka提供专门的后台线程 Log Cleaner 定期地巡检待Compact的主题，看看是否存在满足条件的可删除数据

5.6 Consumer Group 重平衡

- 重平衡 Rebalance本质上是一种协议，规定 Consumer Group下的所有Consumer如何达成一致，来分配订阅Topic的每个分区
- Rebalance的触发条件有3个
  - 组成员数发生变更。比如有新的Consumer实例加入组或者离开组，或是有Consumer实例崩溃被踢出组
  - 订阅主题数发生变更。Consumer Group可以使用正则表达式的方式订阅主题，`consumer.subscribe(Pattern.compile("t.*c"))`就表明该Group订阅所有以字母t开头、字母c结尾的主题，在Consumer Group的运行过程中，你新创建了一个满足这样条件的主题，那么该Group就会发生Rebalance
  - 订阅主题的分区数发生变更。Kafka当前只能允许增加一个主题的分区数，当分区数增加时，就会触发订阅该主题的所有Group开启Rebalance
- Rebalance发生时，Group下所有的Consumer实例都会协调在一起共同参与
- 在Rebalance过程中，所有Consumer实例都会停止消费，等待Rebalance完成

- Coordinator会在什么情况下认为某个Consumer实例已挂从而要退组
  - 当Consumer Group完成Rebalance之后，每个Consumer实例都会定期地向Coordinator发送心跳请求，表明它还存活着
  - 如果某个Consumer实例不能及时地发送这些心跳请求，Coordinator就会认为该Consumer已经死了，从而将其从Group中移除，然后开启新一轮Rebalance
  - Consumer端有个参数，叫session.timeout.ms
  - 该参数的默认值是10秒，即如果Coordinator在10秒之内没有收到Group下某Consumer实例的心跳，它就会认为这个Consumer实例已经挂了
  - Consumer还提供了一个允许你控制发送心跳请求频率的参数，就是heartbeat.interval.ms
  - 这个值设置得越小，Consumer实例发送心跳请求的频率就越高
  - 频繁地发送心跳请求会额外消耗带宽资源，但好处是能够更加快速地知晓当前是否开启Rebalance，因为，目前Coordinator通知各个Consumer实例开启Rebalance的方法，就是将REBALANCE_NEEDED标志封装进心跳请求的响应体中
  - Consumer端还有一个参数，用于控制Consumer实际消费能力对Rebalance的影响，即max.poll.interval.ms，限定了Consumer端应用程序两次调用poll方法的最大时间间隔
  - 默认值是5分钟，表示你的Consumer程序如果在5分钟之内无法消费完poll方法返回的消息，那么Consumer会主动发起离开组的请求，Coordinator也会开启新一轮Rebalance
- 可避免Rebalance的配置
  - 因为未能及时发送心跳，导致Consumer被踢出Group而引发Rebalance
    - 设置**「session.timeout.ms和heartbeat.interval.ms」**的值
    - `session.timeout.ms` = 6s；`heartbeat.interval.ms` = 2s
    - 保证Consumer实例在被判定为dead之前，能够发送至少3轮的心跳请求
  - Consumer消费时间过长引发Rebalance
    - 为业务处理逻辑留下充足的时间，这样Consumer就不会因为处理这些消息的时间太长而引发Rebalance

5.7 生产经验 - 消费者事务

- 如果想完成Consumer端的精准一次性消费，那么需要Kafka消费端将消费过程和提交offset过程做原子绑定
- 此时我们需要将Kafka的offset保存到支持事务的自定义介质（比如MySQL）。这部分知识会在后续项目部分涉及
- 下游消费者必须支持事务，才能做到精确一次性消费

5.8 生产经验 - 数据积压

- 如果是Kafka消费能力不足，则可以考虑增加Topic的分区数，并且同时提升消费组的消费者数量，消费者数 = 分区数
- 如果是下游的数据处理不及时：提高每批次拉取的数量。批次拉取数据过少（拉取数据/处理时间 < 生产速度），使处理的数据小于生产的数据，也会造成数据积压
  - fetch.max.bytes

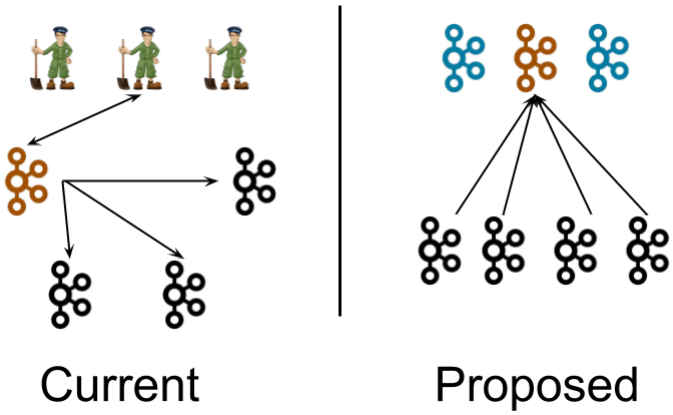
        - 默认Default：52428800（50 m）
        - 消费者获取服务器端一批消息最大的字节数。如果服务器端一批次的数据大于该值（50m）仍然可以拉取回来这批数据，因此，这不是一个绝对最大值
        - 一批次的大小受message.max.bytes （broker config）or max.message.bytes （topic config）影响。
    - max.poll.records
        - 一次poll拉取数据返回消息的最大条数，默认是500条

6. Kafka-Eagle 监控

7. Kafka-Kraft 模式

7.1 Kafka-Kraft架构

- Kafka现有架构，元数据在zookeeper中，运行时动态选举controller，由controller进行Kafka集群管理
- kraft模式架构(实验性), 不再依赖zookeeper集群，而是用三台controller节点代替zookeeper，元数据保存在controller中，由controller直接进行Kafka集群管理



- Kraft好处
    - Kafka不再依赖外部框架，而是能够独立运行
    - controller管理集群时，不再需要从zookeeper中先读取数据，集群性能上升
    - 由于不依赖zookeeper，集群扩展时不再受到zookeeper读写能力限制
    - controller不再动态选举，而是由配置文件规定
        - 这样我们可以有针对性的加强controller节点的配置，而不是像以前一样对随机controller节点的高负载束手无策

7.2 Kafka-Kraft集群部署

- 重新解压一份kafka安装包
- 在hadoop102上修改/opt/module/kafka2/config/kraft/server.properties配置文件

```
# kafka的角色（controller相当于主机、broker节点相当于从机，主机类似zk功能）
process.roles=broker, controller
# 节点ID
node.id=2
# controller服务协议别名
controller.listener.names=CONTROLLER
# 全Controller列表
controller.quorum.voters=2@hadoop102:9093,3@hadoop103:9093,4@hadoop104:9093
#不同服务器绑定的端口
listeners=PLAINTEXT://:9092,CONTROLLER://:9093
#broker服务协议别名
inter.broker.listener.name=PLAINTEXT
#broker对外暴露的地址
advertised.Listeners=PLAINTEXT://hadoop102:9092
#协议别名到安全协议的映射
listener.security.protocol.map=CONTROLLER:PLAINTEXT,PLAINTEXT:PLAINTEXT,SSL:SSL,
SASL_PLAINTEXT:SASL_PLAINTEXT,SASL_SSL:SASL_SSL
# kafka数据存储目录
log.dirs=/opt/module/kafka2/data
```

- 分发kafka
  - `xsync kafka2`
  - 在其他服务器上需要对node.id相应改变，值需要和controller.quorum.voters对应
  - 需要根据各自的主机名称，修改相应的advertised.Listeners地址
- 初始化集群数据目录
  - 首先生成存储目录唯一ID `bin/kafka-storage.sh random-uuid`
  - 用该ID格式化kafka存储目录 每个服务器: `bin/kafka-storage.sh format -t [uuid] -c`
- 启动kafka集群
  - 对于每个服务器 `bin/kafka-server-start.sh -daemon config/kraft/server.properties`
- 停止kafka集群
  - 对于每个服务器 `bin/kafka-server-stop.sh`

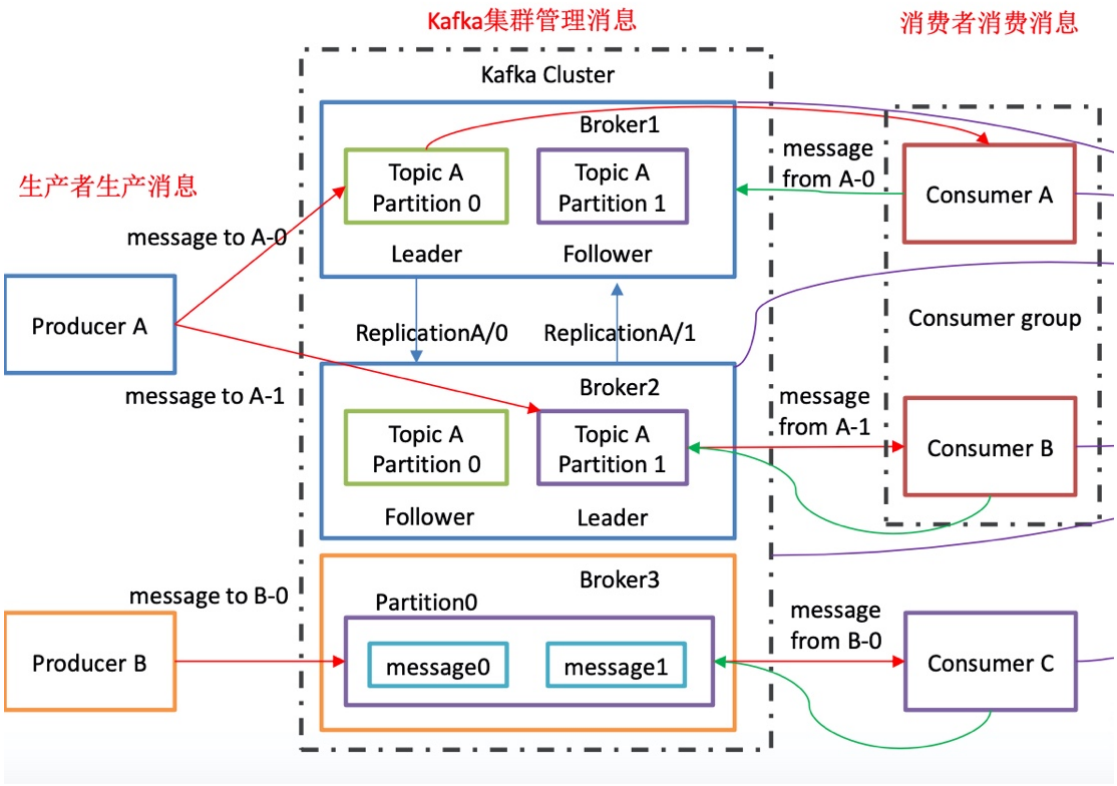
8. 知识梳理

Kafka 基本概念

- Apache Kafka is a **publish-subscribe messaging application** developed by Apache and **written in Scala** programming language. It is an open-source distributed, partitioned and replicated log service and a **message broker application**.
  - The design pattern of Kafka is mainly based on the design of the transactional log
  - 分布式的，基于发布/订阅的消息系统，旨在提供统一的、高吞吐量、低延迟的平台来处理实时数据流
- Kafka 中 message 是以 topic 进行分类消息流，生产者面向topic生产消息，消费者面向topic订阅消费消息
  - Producers issuing communications and publishing messages to a specific Kafka topic;
  - Consume subscribing a topic and also read and process messages from the topic
- Topic 是逻辑上的概念
  - A collection of messages that belong to the same type
  - 发布订阅的对象是主题（Topic），可以为每个业务、每个应用甚至是每类数据都创建专属的主题
- partition 是物理上的概念
  - 一个 topic 可以分为多个 partition，每个 partition 是一个有序的队列/消息日志
  - 每个 partition 对应于一个 log 文件，存储的就是 producer 生产的数据
  - 生产者向分区写入消息，每条消息在分区中的位置信息叫 offset 位移
  - Each record in a partition is assigned and attributed with a unique offset. Multiple partition logs are possible in a single topic. Because of this facility, several users can read from the same topic at the same time
- Producer 负责将记录分配到topic的哪一个 partition中
  - 可以使用循环的方式来简单地实现负载均衡，也可以根据某些语义分区函数(例如：记录中的key)来完成
  - Producer 生产的数据会被不断追加到该 log 文件末端，且每条数据都有自己的 offset
- Consumer Group (CG) 消费者组，由多个 consumer 组成
  - 消费者组内每个消费者负责消费订阅topic的不同分区，即不同消费者不能消费相同分区;
  - 消费者组之间互不影响;
  - 所有的消费者都属于某个消费者组，即一个消费者组是逻辑上的一个订阅者
  - Comsumer Group中的每个comsumer，都会实时记录自己消费到了哪个 offset，以便出错恢复时，从上次的位置继续消费
- Coordinator 协调者
  - 协调者专门为Consumer Group服务，负责为Group执行Rebalance以及提供位移管理和组成员管理等
  - Consumer端应用程序在提交位移时，其实是向Coordinator所在的Broker提交位移，同样地，当Consumer应用启动时，也是向Coordinator所在的Broker发送各种请求，然后由Coordinator负责执行消费者组的注册、成员管理记录等元数据管理操作
  - 所有Broker在启动时，都会创建和开启相应的Coordinator组件

    - Consumer Group如何确定为它服务的Coordinator在哪台Broker上 - 通过Kafka内部主题__consumer_offsets
        - 第1步：确定由__consumer_offsets主题的哪个分区来保存该Group数据：

          `partitionId=Math.abs(groupId.hashCode() % offsetsTopicPartitionCount)`
        - 第2步：找出该分区Leader副本所在的Broker，该Broker即为对应的Coordinator
- Broker 代理
    - Kafka 集群由多个 Broker 组成，Broker 负责接收和处理客户端发送过来的请求，以及对消息进行持久化
    - 一台 kafka 服务器就是一个 Broker , 一个 cluster 由多个 Broker 组成, 一个 Broker 可以容纳多个 topic
    - 为了实现扩展性，一个非常大的 topic 可以分布到多个 Broker (即服务器) 上
- Replica 副本机制
    - 副本，为保证集群中的某个节点发生故障时，该节点上的 partition 数据不丢失，且 kafka 仍然能够继续工作，kafka 提供了副本机制，一个 topic 的每个分区都有若干个副本，分布在不同broker上， 一个 leader 和若干个 follower
- leader: 每个分区多个副本的“主”
    - 生产者发送数据的对象，以及消费者消费数据的对象都是 leader
    - responsible for executing all data read and write commands; the followers have to replicate the process
- follower: 每个分区多个副本中的“从”
    - 实时从 leader 中同步数据，保持和 leader 数据的同步
    - when leader down，one of the followers takes the place and responsibility of the leaders and makes the system stable and helps in the server's load balancing



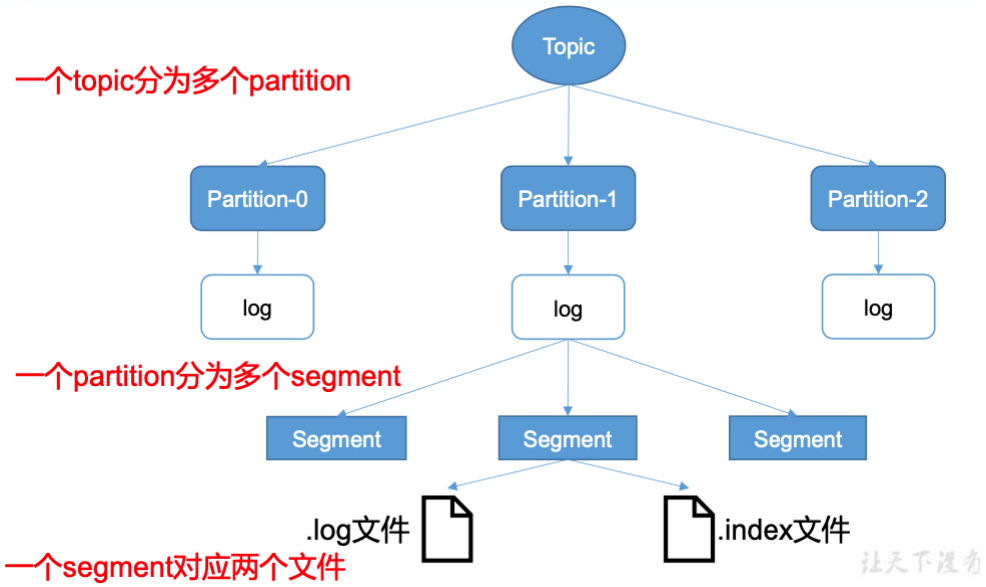
- AR（Assigned Replicas）
  - 分区中的所有副本统称为AR
  - 所有消息会先发送到leader副本，然后follower副本才能从leader中拉取消息进行同步
  - 同步期间，follower对于leader而言会有一定程度的滞后，这个时候follower和leader并非完全同步状态
- ISR（In Sync Replicas）
  - AR中的一个子集，ISR中的副本都是与leader保持完全同步的副本
  - 如果某个在ISR中的follower副本落后于leader副本太多，则会被从ISR中移除，否则如果完全同步，会从OSR中移至ISR集合
  - 默认情况下，当leader副本发生故障时，只有在ISR集合中的follower副本才有资格被选举为新leader，OSR中的副本没有机会
- OSR（Out Sync Replicas）
  - follower副本与leader副本没有完全同步或滞后的副本集合
- HW（High Watermark）
  - 高水位，它标识了一个特定的消息偏移量（offset），消费者只能拉取到这个水位 offset 之前的消息
  - ISR 集合中最小的 LEO 即为分区的 HW，对消费者而言只能消费 HW 之前的消息
- LEO（Log End Offset）
  - 标识当前日志文件中下一条待写入的消息的offset

Kafka 系统架构

- kafka设计思想: 最基本的架构是生产者发布一个消息到Kafka的一个Topic，该Topic的消息存放于的Broker中，消费者订阅这个Topic，然后从Broker中消费消息
- 消息状态: 在Kafka中，消息是否被消费的状态保存在Consumer中，Broker不会关心消息是否被消费或被谁消费，Consumer会记录一个offset值（指向partition中下一条将要被消费的消息位置），如果offset被错误设置可能导致同一条消息被多次消费或者消息丢失
- 消息持久化: Kafka会把消息持久化到本地文件系统中，并且具有极高的性能
- 批量发送: Kafka支持以消息集合为单位进行批量发送，以提高效率
- Push-and-Pull: Kafka中的Producer和Consumer采用的是Push-and-Pull模式，即Producer向Broker Push消息，Consumer从Broker Pull消息
- 分区机制（Partition）: Kafka的Broker端支持消息分区，Producer可以决定把消息发到哪个Partition，在一个Partition中消息的顺序就是Producer发送消息的顺序，一个Topic中的Partition数是可配置的，Partition是Kafka高吞吐量的重要保证
- 系统架构
    - 一个kafka体系架构包括 **「多个Producer」、「多个Consumer」、「多个broker」以及「一个Zookeeper集群」**
    - Producer：生产者，负责将消息发送到kafka中
    - Consumer：消费者，负责从kafka中拉取消息进行消费。
    - Broker：Kafka服务节点，一个或多个Broker组成了一个Kafka集群
    - Zookeeper集群：负责管理kafka集群元数据以及控制器选举等

Kafka 存储机制

- 一个 topic 分为多个 partition, 一个partition对应一个log
- 每个partition采取LogSegment概念, 分为多个LogSegment(文件名包含offset)
- 每个segment对应两个文件: ".index"和".log"文件, 分别表示为索引文件和数据文件
    - .log 数据文件存储消息数据
    - .index 索引文件存储索引信息
    - .timeindex 时间戳索引文件
- log 数据文件大小均相等，超出该设定的阈值大小，将会创建一组新的日志数据和索引文件
- index 索引文件中的元数据指向对应数据文件中message的物理偏移地址
    - 查找数据时，先用二分搜索文件列表，定位到具体的segment
    - 在某个segment的index文件中，二分查找小于等于目标offset的最大索引（此处是**稀疏索引**，只有部分offset的索引），然后从该索引指向的log文件往下找，直到目标offset



Kafka 高性能

- 分区机制
  - 利用Partition实现并行处理：将分区分配到整个集群中，负载均衡，增加`并行操作`的能力
- 顺序读写
  - kafka的消息是不断追加到文件中的，这个特性使kafka可以充分利用磁盘的顺序读写性能
  - 顺序读写不需要硬盘磁头的寻道时间，只需很少的扇区旋转时间，所以速度远快于随机读写
  - Kafka 可以配置异步刷盘，不开启同步刷盘，异步刷盘不需要等写入磁盘后返回消息投递的 ACK，所以它提高了消息发送的吞吐量，降低了请求的延时
- 零拷贝 Zero-Copy
  - 传统的 IO 流程，需要先把数据拷贝到**内核缓冲区**，再从内核缓冲拷贝到用户空间，应用程序处理完成以后，再拷贝回内核缓冲区，这个过程中发生了多次数据拷贝
  - 为了减少不必要的拷贝，Kafka 依赖 Linux 内核提供的 Sendfile 系统调用
  - 数据在内核缓冲区完成输入和输出，不需要拷贝到用户空间处理，这也就避免了重复的数据拷贝
  - Kafka 把所有的消息都存放在单独的文件里，在消息投递时直接通过 Sendfile 方法发送文件，减少了上下文切换，因此大大提高了性能
- MMAP技术
  - 除了 `Sendfile` 之外，另一种零拷贝的实现技术，即 Memory Mapped Files
  - Kafka 使用 Memory Mapped Files 完成内存映射，Memory Mapped Files 对文件的操作不是 write/read，而是直接对内存地址的操作，如果是调用文件的 read 操作，则把数据先读取到内核空间中，然后再复制到用户空间，但 MMAP可以将文件直接映射到用户态的内存空间，省去了用户空间到内核空间复制的开销
  - Producer生产的数据持久化到broker，采用mmap文件映射，将磁盘文件映射到内存, 用户通过修改内存就能修改磁盘文件
  - Cosumer从broker读取数据，采用sendfile，将磁盘文件读到OS内核缓冲区后，根据socket buffer中的位置和偏移量, **直接将socket buffer的数据copy到网卡设备中**
- 批量发送读取

    - Kafka 的批量包括批量写入、批量发布等。它在消息投递时会将消息缓存起来，然后批量发送
    - 消费端在消费消息时，也不是一条一条处理的，而是批量进行拉取，提高了消息的处理速度
- 数据压缩
    - Kafka还支持对消息集合进行压缩，`Producer`可以通过`GZIP`或`Snappy`格式对消息集合进行压缩
    - 好处就是减少传输的数据量，减轻对网络传输的压力
    - Producer压缩之后，在Consumer需进行解压，虽然增加了CPU的工作，但在对大数据处理上，瓶颈在网络上而不是CPU，所以这个成本很值得
- 页缓存 Page-Cache
    - Broker收到数据后，写磁盘时只是将数据写入Page Cache
    - 消费者读也是先去page cache读；但if宕机，cache中数据丢失（可以通过replication机制解决）

Kafka 优缺点

- 优点
    - 吞吐量大 high throughput
    - 副本容错 fault tolerance
    - 持久性 persistent
    - 可扩展性 Scalability
    - 生产者幂等性
- 缺点
    - 无法弹性扩容：对partition的读写都在partition leader所在的broker，如果该broker压力过大，也无法通过新增broker来解决问题
    - 扩容成本高：集群中新增的broker只会处理新topic，如果要分担老topic-partition的压力，需要手动迁移partition，这时会占用大量集群带宽
    - 消费者新加入和退出会造成整个消费组rebalance

Kafka 生产和消费流程

- 从kafka1.0以后默认异步发送，将消息放入后台队列中，然后由单独线程去从队列中取出消息然后发送
    - producer 先从 ZooKeeper 的 “/brokers/…/state” 节点找到该 partition 的leader
    - producer 将消息发送给该 leader
    - leader 将消息写入本地 log
    - followers 从 leader pull 消息，写入本地 log 后 leader 发送 ACK (ack=0/1/-1)
    - leader 收到 ISR 中的 replica 的 ACK 后, 增加 HW(high watermark, 最后 commit 的 offset) 并向 producer 发送 ACK
- producer发送消息时具体到topic的哪一个partition分区，提供了三种方式
    - 指定分区
    - 不指定分区，有指定key 则根据key的hash值与分区数进行运算后确定发送到哪个partition分区
    - 不指定分区，不指定key，则轮询各分区发送

- 消费者一直在轮询，直到有消息
  - 根据offset去查消息
  - 可以指定poll()的timeout时长，若没有数据，则等待一段时间，函数才返回
  - Kafka在消费的过程中向Kafka自带的topic(__consumer_offsets)进行偏移量提交

Kafka Producer Ack机制

- ack=0, producer发送一次就不再发送了，不管是否发送成功
- ack=1, producer只要收到一个leader成功写入的通知就认为推送消息成功了
- ack=-1, 等待所有ISR列表中的follower返回结果，ISR是Broker维护的一个"可靠的follower列表"，配合参数min.insync.replicas使用
- ack的默认值就是1, 是吞吐量与可靠性的一个折中方案

Exactly Once 语义

- 幂等性(idempotent): producer保证发送单个分区的消息只会持久化一条，不会出现重复消息(producerID+SequenceId)
- 为了实现幂等性，它在底层设计架构中引入了ProducerID 和 SequenceNumber
  - ProducerID：在每个新的Producer初始化时，会被分配一个唯一的ProducerID，这个ProducerID对客户端使用者是不可见的
  - SequenceNumber：对于每个ProducerID，Producer发送数据的每个<Topic,Partition>都对应一个从0开始单调递增的SequenceNumber值。Broker端在缓存中保存了这seq number
- 对于接收的每条消息,如果其序号比Broker缓存中序号大于1则接受它,否则将其丢弃,这样就可以实现了避免消息重复提交了.但是只能保证单个Producer对于同一个<Topic,Partition>的Exactly Once语义
- **At Least Once + 幂等性 = Exactly Once**
- To get exactly-once messaging during data production from Kafka, we must **avoid duplicates during data consumption** and **avoid duplication during data production**.
  - In the message, include a primary key (UUID or something) and de-duplicate on the consumer.
  - Whenever get a network error, you should check the last message in that partition to see if your last write succeeded.

Kafka 事务

- 事务(transaction): 保证原子性地将多个消息写入到多个topic/分区，要么全部成功，要么全部回滚
  - 一旦Producer开始发送消息, Transaction Coordinator会将该<Transaction, Topic, Partition>存于Transaction Log内, 并将其状态置为BEGIN
  - 在Producer执行commitTransaction/abortTransaction时，Transaction Coordinator会执行两阶段提交
  - 第一阶段，将Transaction Log内的该事务状态设置为PREPARE_COMMIT或PREPARE_ABORT

    - 第二阶段，将Transaction Marker写入该事务涉及到的所有消息（即将消息标记为committed或aborted）。这一步骤Transaction Coordinator会将请求发送给当前事务涉及到的每个<Topic, Partition>的Leader，将对应的Transaction Marker控制信息写入日志
    - 一旦Transaction Marker写入完成，Transaction Coordinator会将最终的COMPLETE_COMMIT或COMPLETE_ABORT状态写入Transaction Log中以标明该事务结束
- 流处理EOS：流处理本质上可看成是“读取-处理-写入”的管道
    - 此EOS保证整个过程的操作是原子性
    - 注意，这只适用于Kafka Streams

副本同步机制,ISR,LEO,HW

- ISR (**In-Sync Replicas**)
    - 每个分区都有自己的一个 ISR 集合
    - ISR同步副本，里面存放的是和Leader保持同步的副本并含有Leader
    - 这是一个动态调整的集合，当副本由同步变为滞后时会从集合中剔除，而当副本由滞后变为同步时又会加入到集合中
    - 用replica.lag.time.max.ms 参数决定“允许 follower 副本不同步消息的最大时间值”，默认10s；只要在 replica.lag.time.max.ms 时间内 follower 有同步消息，即认为该 follower 处于 ISR 中
    - leader宕机后选举新leader只在ISR中选举
- 同步机制底层原理
    - LEO (Last End Offset) 记录了日志的下一条消息偏移量，即当前最新消息的偏移量加一
    - HW（High Watermark）界定了消费者可见的消息, 最小的LEO
    - 在每个副本中都存有LEO和HW，而Leader副本中除了存有自身的LEO和HW，还存储了其他Follower副本的LEO和HW值
    - Leader更新LEO和HW时机有两个：收到生产者的消息、被Follower拉取消息
        - 当收到 Producer 消息时，会用当前偏移量加1来更新LEO，然后取LEO和远程ISR副本中LEO的最小值更新HW
        - LEO = CurrentOffset + 1；HW = min(LEO, RemoteIsrLEO)
        - 当 Follower 拉取消息时，会更新Leader上存储的Follower副本LEO，然后判断是否需要更新HW，更新的方式和上述相同
        - RemoteLEO = FollowerLEO；HW = min(LEO, RemoteIsrLEO)
    - Follower更新LEO和HW的时机只有向Leader拉取了消息之后
        - 会用当前的偏移量加1来更新LEO，并且用Leader的HW值和当前LEO的最小值来更新HW
        - LEO = CurrentOffset + 1；HW = min(LEO, LeaderHW)
- 除了这两种正常情况，当发生故障时，例如Leader宕机，Follower被选为新的Leader，会尝试更新HW。还有副本被踢出ISR时，也会尝试更新HW

Load Balancing

- The load balancing process spreads out the message load between partitions while preserving message ordering. Kafka enables users to specify the exact partition for a message.
- leaders perform the task of all read and write requests for the partition. On the other hand, followers passively replicate the leader. At the time of leader failure, one of the followers takes over the role of the leader, and this entire process ensures load balancing of the servers.

Retention period

- Within the Kafka cluster, the retention period is used to retain all the published records without checking whether they have been consumed or not.
- Using a configuration setting for the retention period, we can easily discard the records.
- The main purpose of discarding the records from the Kafka cluster is to free up some space.

Kafka 消息丢失问题

- producer 丢失数据
  - Producer是异步发送消息，如果调用的是 `producer.send(msg)` 这个API，那么它通常会立即返回，但此时不能认为消息发送已成功完成
  - 丢数据场景: kafka会合并请求在本地buffer中打包异步发送，producer挂的情况下本地buffer中没来得及发送的数据就丢了。如果这里设置发送方式为同步发送，producer.type= sync，但是性能会大大下降，不考虑
  - 解决方案: 使用带有回调通知的 `producer.send(msg, callback)` 设置retries，通过重试机制重新发送消息，避免消息丢失
- broker
  - 丢数据场景: kafka将数据异步批量的存储在磁盘中。为了提高性能，减少刷盘次数，kafka采用了批量刷盘的做法，数据暂存在PageCache。在刷盘的间隔中断电，造成数据丢失
  - 解决方案: 设置参数 `replication.factor >= 3`（每个partition三个副本）通过消息冗余来防止消息丢失；
  - 设置 `min.insync.replicas > 1`，控制的是消息至少要被写入到多少个副本才算是已提交，设置成大于1可以提升消息持久性，在实际环境中千万不要使用默认值1
  - 确保 `replication.factor > min.insync.replicas`，如果二者相等的话，只要有一个副本挂掉，那么整个分区就无法工作了，建议设置为 `replication.factor = min.insync.replicas + 1`
  - 设置 `unclean.leader.election.enable = false`，控制的是哪些Broker有资格竞选分区的Leader，如果一个Broker落后原先的Leader太多，那么它一旦成为新的Leader，必然会造成消息的丢失，故一般都要将该参数设置成false，不允许这种情况的发生
- comuser 丢失数据
  - 丢数据场景: Consumer自动提交offset和消费消息是异步的，如果消费过程未成功（比如抛出异常），而offset已经提交了，此时消息就丢失了

- 解决方案: 设置 `enable.auto.commit = false`, 在确保消息消费完成后手动commit，可以保证消息至少被消费一次

Kafka 重复消费问题

- 消费重复的场景
  - autocommit 情况下, consumer 在消费过程中，在commit offset之前应用进程被强制kill掉或发生异常退出
- 解决方案
  - 下游消费者需要支持事务，做到精确原子性一次性消费
  - 将消息的唯一标识（如）保存到外部介质中 (redis)，每次消费时判断是否处理过即可
- 消费者消费时间过长
  - max.poll.interval.ms参数定义了两次poll的最大间隔，它的默认值是 5 分钟，如果在5 分钟之内无法消费完 poll 方法返回的消息，那么 Consumer 会主动发起离开组的请求，Coordinator 也会开启新一轮 Rebalance
- 解决方案
  - 提高消费能力，提高单条消息的处理速度；根据实际场景可讲max.poll.interval.ms值设置大一点，避免不必要的rebalance；可适当减小max.poll.records的值，默认值是500，可根据实际消息速率适当调小
  - 生成消息时，可加入唯一标识符 (如用snowflake定义消息体id)，在消费端，每次消费时判断是否处理过去重

Kafka 消息顺序问题

- kafka的 topic 是无序的，但是一个 topic 包含多个 partition ，每个 partition 内部是有序的
- 无法保证全局的消息顺序性的，只能保证主题的某个 partition 的消息顺序性
  - 方案一，可以只设置一个partition分区
  - 方案二，producer将消息发送到指定partition分区
- 通过设置相同key来保证消息有序性，会有一点缺陷
  - 消息发送设置了重试机制，并且异步发送，消息A和B设置相同的key，业务上A先发，B后发，由于网络或者其他原因A发送失败，B发送成功；A由于发送失败就会重试且重试成功，这时候消息顺序B在前A在后，与业务发送顺序不一致
  - 解决这个问题，需要设置参数max.in.flight.requests.per.connection=1，其含义是限制客户端在单个连接上能够发送的未响应请求的个数，设置此值是1表示kafka broker在响应请求之前client不能再向同一个broker发送请求，这个参数默认值是5

重平衡rebalance？如何避免?

- Rebalance 规定了一个 Consumer Group 下的所有 consumer 如何达成一致，来分配订阅 Topic 的每个分区
- 例如：某 Group 下有 20 个 consumer 实例，它订阅了一个具有 100 个 partition 的 Topic
- 正常情况下，kafka 会为每个 Consumer 平均的分配 5 个分区, 分配的过程就是 Rebalance

- Rebalance 的触发条件:
  - 组成员个数发生变化。例如有新的 consumer 实例加入该消费组或者离开组
  - 订阅的 Topic 个数发生变化
  - 订阅 Topic 的分区数发生变化
- 在 Rebalance 的过程中 consumer group 下的所有消费者实例都会停止工作，等待 Rebalance 过程完成
- **尽量避免**：放宽判断consumer挂了的条件，如超时时间
- Kafka Rebalance机制分析

扩容、缩容

- 不管扩容还是缩容，或者是故障后手动补齐分区，实质都是分区重分配，使用kafka-reassign-partitions.sh脚本即可
- 扩容也就是新增节点，扩容后老的数据不会自动迁移，只有新创建的topic才可能会分配到新增的节点上面
  - 如果不需要迁移旧数据，那直接把新的节点启动起来就行了，不需要做额外的操作
  - 有的时候，新增节点后，我们会将一些老数据迁移到新的节点上，以达到负载均衡的目的，这时需要手动操作
  - Kafka提供了一个脚本kafka-reassign-partitions.sh，通过这个脚本可以重新分配分区的分布
- 缩容：某个节点故障了，且无法恢复

Kafka 消息压缩

- kafka的消息层次分为两层：消息集合(message set)以及消息(message)
- 一个消息集合包含若干条日志项(record item)，而日志项才是真正封装消息的地方
- kafka底层的消息日志由一系列消息集合日志项组成
- kafka通常不会直接操作具体的一条条消息，它总是在消息集合这个层面上进行写入操作
- message set = message header + message
  - 抽取了消息的公共部分放到消息集合中；去掉每条消息的公共部分，减少了总体积
  - 消息的CRC校验由对每一条消息，移动到了对消息集合进行校验，减少了校验次数，节省了cpu
- 消息压缩是为了节省空间，但消耗CPU资源
- 实习项目中没开启消息压缩
- 压缩可能会发生在producer端和broker端（对整个消息集合压缩），消费者解压缩（通过消息集合知道压缩算法比如gzip）
- 大部分情况下 Broker 从 Producer 端接收到消息后原封不动地保存，除非producer和broker压缩算法不一致 or broker为了兼容新老版本消息格式而进行新->老的格式转换导致解压和重新压缩

Kafka 和zookeeper的关系

- 注册broker，在/brokers/ids下创建这个broker节点，如/brokers/ids/[0...N]，并保存broker的IP地址和端口。这个节点是临时节点，一旦broker宕机，这个临时节点会被删除
- 注册topic：以/brokers/topics/[topic_name]的形式记录在Zookeeper
- 注册consumer，包括consumer消费的partition
- 选举leader
- 监控每个partition leader存活性

Zk可以做什么

- Zk = 文件系统 + 通知机制
- 配置管理：配置保存到zk目录并监听，一旦配置信息变化就收到zk通知
- 集群监控：机器上下线监控
- 分布式锁

面试题

001. 为什么需要消息系统

- 解耦：允许你独立的扩展或修改两边的处理过程，只要确保它们遵守同样的接口约束。
- 冗余：消息队列把数据进行持久化直到它们已经被完全处理，通过这一方式规避了数据丢失风险。许多消息队列所采用的”插入-获取-删除”范式中，在把一个消息从队列中删除之前，需要你的处理系统明确的指出该消息已经被处理完毕，从而确保你的数据被安全的保存直到你使用完毕。
- 扩展性：因为消息队列解耦了你的处理过程，所以增大消息入队和处理的频率是很容易的，只要另外增加处理过程即可。
- 灵活性 & 峰值处理能力：在访问量剧增的情况下，应用仍然需要继续发挥作用，但是这样的突发流量并不常见。如果为以能处理这类峰值访问为标准来投入资源随时待命无疑是巨大的浪费。使用消息队列能够使关键组件顶住突发的访问压力，而不会因为突发的超负荷的请求而完全崩溃。
- 可恢复性：系统的一部分组件失效时，不会影响到整个系统。消息队列降低了进程间的耦合度，所以即使一个处理消息的进程挂掉，加入队列中的消息仍然可以在系统恢复后被处理。
- 顺序保证：在大多使用场景下，数据处理的顺序都很重要。大部分消息队列本来就是排序的，并且能保证数据会按照特定的顺序来处理。（Kafka 保证一个 Partition 内的消息的有序性）
- 缓冲：有助于控制和优化数据流经过系统的速度，解决生产消息和消费消息的处理速度不一致的情况。
- 异步通信：很多时候，用户不想也不需要立即处理消息。消息队列提供了异步处理机制，允许用户把一个消息放入队列，但并不立即处理它。想向队列中放入多少消息就放多少，然后在需要的时候再去处理它们

002. 如何进行产品选型

- Kafka:
  - 优点： 吞吐量非常大，性能非常好，集群高可用
  - 缺点：会丢数据，功能比较单一
  - 使用场景：日志分析、大数据采集
- RabbitMQ
  - 优点： 消息可靠性高，功能全面
  - 缺点：吞吐量比较低，消息积累会严重影响性能。erlang语言不好定制
  - 使用场景：小规模场景
- RocketMQ
  - 优点：高吞吐、高性能、高可用，功能非常全面
  - 缺点：开源版功能不如云上商业版。官方文档和周边生态还不够成熟。客户端只支持java
  - 使用场景：几乎是全场景

003. 消息传输传统方法

- **Queuing:**
  - In the queuing method, a pool of consumers may read messages from the server, and each message goes to one of them.
- **Publish-Subscribe:**
  - In the Publish-Subscribe model, messages are broadcasted to all consumers.

RabbitMQ

001. 简述RabbitMQ的架构设计

- Broker：rabbitmq的服务节点
- Queue：队列，是RabbitMQ的内部对象，用于存储消息
  - RabbitMQ中消息只能存储在队列中。生产者投递消息到队列，消费者从队列中获取消息并消费。多个消费者可以订阅同一个队列，这时队列中的消息会被平均分摊(轮询)给多个消费者进行消费，而不是每个消费者都收到所有的消息进行消费。(注意：RabbitMQ不支持队列层面的广播消费，如果需要广播消费，可以采用一个交换器通过路由Key绑定多个队列，由多个消费者来订阅这些队列的方式
- Exchange：交换器。生产者将消息发送到Exchange，由交换器将消息路由到一个或多个队列中。如果路由不到，或返回给生产者，或直接丢弃，或做其它处理
- RoutingKey：路由Key。生产者将消息发送给交换器的时候，一般会指定一个RoutingKey，用来指定这个消息的路由规则。这个路由Key需要与交换器类型和绑定键(BindingKey)联合使用才能最终生效。在交换器类型和绑定键固定的情况下，生产者可以在发送消息给交换器时通过指定RoutingKey来决定消息流向哪里
- Binding：通过绑定将交换器和队列关联起来，在绑定的时候一般会指定一个绑定键，这样RabbitMQ就可以指定如何正确的路由到队列了

    - 交换器和队列实际上是多对多关系。就像关系数据库中的两张表。他们通过BindingKey做关联(多对多关系表)。在投递消息时，可以通过Exchange和RoutingKey(对应BindingKey)就可以找到相对应的队列
- 信道：信道是建立在Connection 之上的虚拟连接。当应用程序与Rabbit Broker建立TCP连接的时候，客户端紧接着可以创建一个AMQP 信道(Channel)，每个信道都会被指派一个唯一的D。RabbitMQ 处理的每条AMQP 指令都是通过信道完成的。信道就像电缆里的光纤束。一条电缆内含有许多光纤束，允许所有的连接通过多条光线束进行传输和接收

002. RabbitMQ如何确保消息发送接收

- 发送方确认机制：
    - 信道需要设置为 confirm 模式，则所有在信道上发布的消息都会分配一个唯一 ID。一旦消息被投递到queue（可持久化的消息需要写入磁盘），信道会发送一个确认给生产者（包含消息唯一 ID）。如果 RabbitMQ 发生内部错误从而导致消息丢失，会发送一条 nack（未确认）消息给生产者。所有被发送的消息都将被 confirm（即 ack） 或者被nack一次。但是没有对消息被 confirm 的快慢做任何保证，并且同一条消息不会既被 confirm又被nack
    - 发送方确认模式是异步的，生产者应用程序在等待确认的同时，可以继续发送消息。当确认消息到达生产者，生产者的回调方法会被触发。
    - ConfirmCallback接口：只确认是否正确到达 Exchange 中，成功到达则回调
    - ReturnCallback接口：消息失败返回时回调
- 接收方确认机制：
    - 消费者在声明队列时，可以指定noAck参数，当noAck=false时，RabbitMQ会等待消费者显式发回ack信号后才从内存(或者磁盘，持久化消息)中移去消息。否则，消息被消费后会被立即删除
    - 消费者接收每一条消息后都必须进行确认（消息接收和消息确认是两个不同操作）。只有消费者确认了消息，RabbitMQ 才能安全地把消息从队列中删除
    - RabbitMQ不会为未ack的消息设置超时时间，它判断此消息是否需要重新投递给消费者的唯一依据是消费该消息的消费者连接是否已经断开。这么设计的原因是RabbitMQ允许消费者消费一条消息的时间可以很长。保证数据的最终一致性
    - 如果消费者返回ack之前断开了链接，RabbitMQ 会重新分发给下一个订阅的消费者。（可能存在消息重复消费的隐患，需要去重）

003. RabbitMQ事务消息

- 通过对信道的设置实现
    - channel.txSelect()；通知服务器开启事务模式；服务端会返回Tx.Select-Ok
    - channel.basicPublish；发送消息，可以是多条，可以是消费消息提交ack
    - channel.txCommit()提交事务；
    - channel.txRollback()回滚事务；
- 消费者使用事务：
    - autoAck=false，手动提交ack，以事务提交或回滚为准；
    - autoAck=true，不支持事务的，也就是说你即使在收到消息之后在回滚事务也是于事无补的，队列已经把消息移除了

- 如果其中任意一个环节出现问题，就会抛出IoException异常，用户可以拦截异常进行事务回滚，或决定要不要重复消息。
- 事务消息会降低rabbitmq的性能

004. RabbitMQ死信队列、延时队列

- 条件
  - 消息被消费方否定确认，使用 channel.basicNack 或 channel.basicReject ，并且此时requeue属性被设置为 false
  - 消息在队列的存活时间超过设置的TTL时间
  - 消息队列的消息数量已经超过最大队列长度
- 那么该消息将成为“死信” (Dead-Letter)。“死信”消息会被RabbitMQ进行特殊处理，如果配置了死信队列 (Dead-Letter-Queue)，那么该消息将会被丢进死信队列中，如果没有配置，则该消息将会被丢弃
- 为每个需要使用死信的业务队列配置一个死信交换机，这里同一个项目的死信交换机可以共用一个，然后为每个业务队列分配一个单独的路由key，死信队列只不过是绑定在死信交换机上的队列，死信交换机也不是什么特殊的交换机，只不过是用来接受死信的交换机，所以可以为任何类型【Direct、Fanout、Topic】
- TTL：一条消息或者该队列中的所有消息的最大存活时间
- 如果一条消息设置了TTL属性或者进入了设置TTL属性的队列，那么这条消息如果在TTL设置的时间内没有被消费，则会成为“死信”。如果同时配置了队列的TTL和消息的TTL，那么较小的那个值将会被使用
- 只需要消费者一直消费死信队列里的消息

005. RabbitMQ镜像队列机制

- 镜像queue有master节点和slave节点。master和slave是针对一个queue而言的，而不是一个node作为所有queue的master，其它node作为slave。一个queue第一次创建的node为它的master节点，其它node为slave节点
- 无论客户端的请求打到master还是slave最终数据都是从master节点获取。当请求打到master节点时，master节点直接将消息返回给client，同时master节点会通过GM（Guaranteed Multicast）协议将queue的最新状态广播到slave节点。GM保证了广播消息的原子性，即要么都更新要么都不更新
- 当请求打到slave节点时，slave节点需要将请求先重定向到master节点，master节点将将消息返回给client，同时master节点会通过GM协议将queue的最新状态广播到slave节点
- 如果有新节点加入，RabbitMQ不会同步之前的历史数据，新节点只会复制该节点加入到集群之后新增的消息

KafkaMQ

001. Kafka是什么

- Kafka 是一种高吞吐量、分布式、基于发布/订阅的消息系统，Scala语言编写，Apache 的开源项目
- broker：Kafka 服务器，负责消息存储和转发
- topic：消息类别， Kafka 按照 topic 来分类消息
- partition：topic 的分区，一个 topic 可以包含多个 partition，topic 消息保存在各个partition 上

- offset：消息在日志中的位置，可以理解是消息在 partition 上的偏移量，也是代表该消息的唯一序号
- Producer：消息生产者
- Consumer：消息消费者
- Consumer Group：消费者分组，每个 Consumer 必须属于一个 group
- Zookeeper：保存着集群 broker、 topic、 partition等 meta 数据；另外，还负责 broker 故障发现，partition leader 选举，负载均衡等功能

002. Kafka为什么吞吐量高

- Kafka的生产者采用的是**异步通信机制**，当发送一条消息时，消息并没有发送到Broker而是缓存起来，然后直接向业务返回成功，当缓存的消息达到一定数量时再批量发送给Broker
- 这种做法减少了网络io，从而提高了消息发送的吞吐量，但是如果消息生产者宕机，会导致消息丢失，业务出错，所以理论上kafka利用此机制提高了性能却降低了可靠性

003. Kafka的Push和Pull分别优缺点

- Push 表示Broker主动给消费者推送消息，有消息时才会推送，但是消费者不能按自己的能力来消费消息，推过来多少消息就得消费多少消息，可能会造成网络堵塞，消费者压力大等问题
- Pull 表示消费者主动拉取，可以批量拉取，也可以单条拉取
    - Pull 可以由消费者自己控制，根据自己的消息处理能力来进行控制
    - 如果broker没有可供消费的消息，将导致consumer不断在循环中轮询，直到新消息到达
- Kafka是Push还是Pull模式
    - customer应该从brokes拉取消息还是brokers将消息推送到consumer?
    - Kafka中的Producer和Consumer采用的是Push-and-Pull模式，Producer向Broker Push消息，Consumer从Broker Pull消息

004. Kafka如何保证高可用

- 高可用性: 系统无间断地执行其功能的能力，代表系统的可用性程度
- 备份机制
    - Kafka允许同一个Partition存在多个消息副本，每个Partition的副本通常由1个Leader及多个Follower组成
    - Kafka会尽量将所有的Partition以及各Partition的副本均匀地分配到整个集群的各个Broker上
- ISR机制
    - ISR 中的副本都是与 Leader 同步的副本，相反，不在 ISR 中的追随者副本就被认为是与 Leader 不同步的
    - 各Partition的Leader负责维护ISR列表并将ISR的变更同步至ZooKeeper，被移出ISR的Follower会继续向Leader发FetchRequest请求，试图再次跟上Leader重新进入ISR
    - ISR中所有副本都跟上了Leader，通常只有ISR里的成员才可能被选为Leader
- Unclean领导者选举
    - 当Kafka中unclean.leader.election.enable配置为true(默认值为false)且ISR中所有副本均宕机的情况下，才允许ISR外的副本被选为Leader，此时会丢失部分已应答的数据

    - 开启 Unclean 领导者选举可能会造成数据丢失，但好处是，它使得分区 Leader 副本一直存在，不至于停止对外提供服务，因此提升了高可用性，反之，禁止 Unclean 领导者选举的好处在于维护了数据的一致性，避免了消息丢失，但牺牲了高可用性
- ACK机制
    - 生产者发送消息中包含acks字段，该字段代表Leader应答生产者前Leader收到的应答数
    - acks=0 无需等待服务端的任何确认；acks=1 Leader 接收到消息就认为成功；acks=all 等待ISR中的所有副本确认后再做出应答
- 故障恢复机制
    - Kafka引入Leader选举及失败恢复机制
    - 首先需要在集群所有Broker中选出一个Controller，负责各Partition的Leader选举以及Replica的重新分配
    - 当出现Leader故障后，Controller会将Leader/Follower的变动通知到需为此作出响应的Broker
    - Kafka使用ZooKeeper存储Broker、Topic等状态数据，Kafka集群中的Controller和Broker会在ZooKeeper指定节点上注册Watcher(事件监听器)，以便在特定事件触发时，由ZooKeeper将事件通知到对应Broker
- Controller
    - 集群中的Controller也会出现故障，因此Kafka让所有Broker都在ZooKeeper的Controller节点上注册一个Watcher
    - Controller发生故障时对应的Controller临时节点会自动删除，此时注册在其上的Watcher会被触发，所有活着的Broker都会去竞选成为新的Controller

010. 为什么要使用 kafka / 消息队列

- 缓冲buffering和削峰clipping： 上游数据时有突发流量，下游可能扛不住，或者下游没有足够多的机器来保证冗余，kafka在中间可以起到一个缓冲的作用，把消息暂存在kafka中，下游服务就可以按照自己的节奏进行慢处理
- 解耦和扩展性： 项目开始的时候，并不能确定具体需求。消息队列可以作为一个接口层，解耦重要的业务流程。只需要遵守约定，针对数据编程即可获取扩展能力
    - 冗余： 可以采用一对多的方式，一个生产者发布消息，可以被多个订阅topic的服务消费到，供多个毫无关联的业务使用
    - 健壮性： 消息队列可以堆积请求，所以消费端业务即使短时间死掉，也不会影响主要业务的正常进行
    - 异步通信： 很多时候，用户不想也不需要立即处理消息。消息队列提供了异步处理机制，允许用户把一个消息放入队列，但并不立即处理它。想向队列中放入多少消息就放多少，然后在需要的时候再去处理它们

011. Kafka中的ISR、AR又代表什么？ISR的伸缩又指什么

- ISR: In-Sync Replicas 副本同步队列

- AR: Assigned Replicas 所有副本ISR是由leader维护，follower从leader同步数据有一些延迟（包括延迟时间replica.lag.time.max.ms和延迟条数replica.lag.max.messages两个维度, 当前最新的版本0.10.x中只支持replica.lag.time.max.ms这个维度），任意一个超过阈值都会把follower剔除出ISR, 存入OSR（Outof-Sync Replicas）列表，新加入的follower也会先存放在OSR中
- AR=ISR+OSR

012. Kafka高效文件存储设计特点

- Kafka 把 topic 中一个 parition 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
- 通过索引信息可以快速定位 message 和确定 response 的最大大小
- 通过 index 元数据全部映射到 memory，可以避免 segment file 的 IO 磁盘操作
- 通过索引文件稀疏存储，可以大幅降低 index 文件元数据占用空间大小

013. Kafka与传统消息系统之间有三个关键区别

- Kafka 持久化日志，这些日志可以被重复读取和无限期保留
- Kafka 是一个分布式系统：它以集群的方式运行，可以灵活伸缩，在内部通过复制数据提升容错能力和高可用性
- Kafka 支持实时的流式处理

014. Kafka创建 Topic 时如何将分区放置到不同的 Broker 中

- 副本因子不能大于 Broker 的个数；
- 第一个分区（编号为 0）的第一个副本放置位置是随机从 brokerList 选择的；
- 其他分区的第一个副本放置位置相对于第 0 个分区依次往后移。也就是如果我们有 5 个Broker，5 个分区，假设第一个分区放在第四个 Broker 上，那么第二个分区将会放在第五个 Broker 上；第三个分区将会放在第一个 Broker 上；第四个分区将会放在第二个Broker 上，依次类推；
- 剩余的副本相对于第一个副本放置位置其实是由 nextReplicaShift 决定的，而这个数也是随机产生的

015. Kafka的消费者如何消费数据

- 消费者每次消费数据的时候，消费者都会记录消费的物理偏移量（ offset）的位置等到下次消费时，他会接着上次位置继续消费

016. Kafka消费者负载均衡策略

- 一个消费者组中的一个分片对应一个消费者成员，他能保证每个消费者成员都能访问，如果组中成员太多会有空闲的成员

017. kafaka生产数据时数据的分组策略

- 生产者决定数据产生到集群的哪个 partition 中每一条消息都是以（ key， value）格式 Key是由生产者发送数据传入所以生产者（ key）决定了数据产生到集群的哪个 partition

018. Kafka中是怎么体现消息顺序性的

- kafka每个partition中的消息在写入时都是有序的，消费时，每个partition只能被每一个group中的一个消费者消费，保证了消费时也是有序的。整个topic不保证有序。如果为了保证topic整个有序，那么将partition调整为1

019. Kafka如何实现延迟队列

- Kafka并没有使用JDK自带的Timer或者DelayQueue来实现延迟的功能，而是基于时间轮自定义了一个用于实现延迟功能的定时器（SystemTimer）。JDK的Timer和DelayQueue插入和删除操作的平均时间复杂度为O(nlog(n))，并不能满足Kafka的高性能要求，而基于时间轮可以将插入和删除操作的时间复杂度都降为O(1)。
- 时间轮的应用并非Kafka独有，其应用场景还有很多，在Netty、Akka、Quartz、Zookeeper等组件中都存在时间轮的踪影。底层使用数组实现，数组中的每个元素可以存放一个TimerTaskList对象。TimerTaskList是一个环形双向链表，在其中的链表项TimerTaskEntry中封装了真正的定时任务TimerTask.
- Kafka中到底是怎么推进时间的呢？Kafka中的定时器借助了JDK中的DelayQueue来协助推进时间轮。具体做法是对于每个使用到的TimerTaskList都会加入到DelayQueue中。Kafka中的TimingWheel专门用来执行插入和删除TimerTaskEntry的操作，而DelayQueue专门负责时间推进的任务。再试想一下，DelayQueue中的第一个超时任务列表的expiration为200ms，第二个超时任务为840ms，这里获取DelayQueue的队头只需要O(1)的时间复杂度。如果采用每秒定时推进，那么获取到第一个超时的任务列表时执行的200次推进中有199次属于“空推进”，而获取到第二个超时任务时有需要执行639次“空推进”，这样会无故空耗机器的性能资源，这里采用DelayQueue来辅助以少量空间换时间，从而做到了“精准推进”。
- Kafka中的定时器真可谓是“知人善用”，用TimingWheel做最擅长的任务添加和删除操作，而用DelayQueue做最擅长的时间推进工作，相辅相成。

Kafka 和 RabbitMQ 区别

- RabbitMQ 采用push的方式；Kafka 采用pull的方式
- Kafka 可以保证按顺序处理消息；RabbitMQ在这块就相对比较弱
- RabbitMQ 在金融场景中经常使用，具有较高的严谨性，数据丢失的可能性更小，同时具备更高的实时性；而Kafka 优势主要体现在吞吐量上，虽然可以通过策略实现数据不丢失，但从严谨性角度来讲，大不如RabbitMQ

RocketMQ

020. RocketMQ的事务消息是如何实现的

- 生产者订单系统先发送一条half消息到Broker，half消息对消费者而言是不可见的
- 再创建订单，根据创建订单成功与否，向Broker发送commit或rollback
- 并且生产者订单系统还可以提供Broker回调接口，当Broker发现一段时间half消息没有收到任何操作命令，则会主动调此接口来查询订单是否创建成功
- 一旦half消息commit了，消费者库存系统就会来消费，如果消费成功，则消息销毁，分布式事务成功结束

- 如果消费失败，则根据重试策略进行重试，最后还失败则进入死信队列，等待进一步处理

021. 为什么RocketMQ不使用Zookeeper作为注册中心

- 根据CAP理论，同时最多只能满足两个点，而zookeeper满足的是CP，也就是说zookeeper并不能保证服务的可用性，zookeeper在进行选举的时候，整个选举的时间太长，期间整个集群都处于不可用的状态，而这对于一个注册中心来说肯定是不能接受的，作为服务发现来说就应该是为可用性而设计。

  基于性能的考虑，NameServer本身的实现非常轻量，而且可以通过增加机器的方式水平扩展，增加集群的抗压能力，而zookeeper的写是不可扩展的，而zookeeper要解决这个问题只能通过划分领域，划分多个zookeeper集群来解决，首先操作起来太复杂，其次这样还是又违反了CAP中的A的设计，导致服务之间是不连通的。

  持久化的机制来带的问题，ZooKeeper 的 ZAB 协议对每一个写请求，会在每个 ZooKeeper 节点上保持写一个事务日志，同时再加上定期的将内存数据镜像（

  Snapshot）到磁盘来保证数据的一致性和持久

  性，而对于一个简单的服务发现的场景来说，这其实没有太大的必要，这个实现方案太重了。而且本身存储的数据应该是高度定制化的。

  消息发送应该弱依赖注册中心，而RocketMQ的设计理念也正是基于此，生产者在第一次发送消息的时候从NameServer获取到Broker地址后缓存到本地，如果NameServer整个集群不可用，短时间内对于生

  产者和消费者并不会产生太大影响

022. RocketMQ的实现原理

- RocketMQ由NameServer注册中心集群、Producer生产者集群、Consumer消费者集群和若干Broker（RocketMQ进程）组成，它的架构原理是这样的：
  - Broker在启动的时候去向所有的NameServer注册，并保持长连接，每30s发送一次心跳
  - Producer在发送消息的时候从NameServer获取Broker服务器地址，根据负载均衡算法选择一台服务器来发送消息
  - Conusmer消费消息的时候同样从NameServer获取Broker地址，然后主动拉取消息来消费

023. RocketMQ为什么速度快

- 因为使用了顺序存储、Page Cache和异步刷盘
- 我们在写入commitlog的时候是顺序写入的，这样比随机写入的性能就会提高很多，写入commitlog的时候并不是直接写入磁盘，而是先写入操作系统的PageCache，最后由操作系统异步将缓存中的数据刷到磁盘

024. 消息队列如何保证消息可靠传输

- 消息可靠传输代表了两层意思，既不能多也不能少
  - 为了保证消息不多，也就是消息不能重复，也就是生产者不能重复生产消息，或者消费者不能重复消费消息
  - 首先要确保消息不多发，这个不常出现，也比较难控制，因为如果出现了多发，很大的原因是生产者自己的原因，如果要避免出现问题，就需要在消费端做控制
  - 要避免不重复消费，最保险的机制就是消费者实现幂等性，保证就算重复消费，也不会有问题，通过幂等性，也能解决生产者重复发送消息的问题
  - 消息不能少，意思就是消息不能丢失，生产者发送的消息，消费者一定要能消费到，对于这个问题，就要考虑两个方面
  - 生产者发送消息时，要确认broker确实收到并持久化了这条消息，比如RabbitMQ的confirm机制，Kafka的ack机制都可以保证生产者能正确的将消息发送给broker
  - broker要等待消费者真正确认消费到了消息时才删除掉消息，这里通常就是消费端ack机制，消费者接收到一条消息后，如果确认没问题了，就可以给broker发送一个ack，broker接收到ack后才会删除消息

025. 消息队列有哪些作用

- 解耦：使用消息队列来作为两个系统之间的通讯方式，两个系统不需要相互依赖了
- 异步：系统A给消息队列发送完消息之后，就可以继续做其他事情了
- 流量削峰：如果使用消息队列的方式来调用某个系统，那么消息将在队列中排队，由消费者自己控制消费速度

026. 死信队列是什么？延时队列是什么?

- 死信队列也是一个消息队列，它是用来存放那些没有成功消费的消息的，通常可以用来作为消息重试
- 延时队列就是用来存放需要在指定时间被处理的元素的队列，通常可以用来处理一些具有过期性操作的业务，比如十分钟内未支付则取消订单

027. 如何保证消息的高效读写?

- 零拷贝： kafka和RocketMQ都是通过零拷贝技术来优化文件读写
- 传统文件复制方式： 需要对文件在内存中进行四次拷贝。
- 零拷贝： 有两种方式， mmap和transfile，Java当中对零拷贝进行了封装
  - Mmap方式通过MappedByteBuffer对象进行操作; 而transfile通过FileChannel来进行操作
  - Mmap 适合比较小的文件，通常文件大小不要超过1.5G ~2G 之间; Transfile没有文件大小限制
- RocketMQ当中使用Mmap方式来对文件进行读写
- 在kafka当中，他的index日志文件也是通过mmap的方式来读写的。在其他日志文件当中，并没有使用零拷贝的方式。Kafka使用transfile方式将硬盘数据加载到网卡

028. 如何设计一个MQ

- 实现一个单机的队列数据结构， 高效、可扩展
- 将单机队列扩展成为分布式队列 - 分布式集群管理
- 基于Topic定制消息路由策略。- 发送者路由策略，消费者与队列对应关系，消费者路由策略
- 实现高效的网络通信。- Netty Http
- 规划日志文件，实现文件高效读写。- 零拷贝，顺序写。 服务重启后，快速还原运行现场。
- 定制高级功能，死信队列、延迟队列、事务消息等等。 - 贴合实际，随意发挥。