时间都哪去了
码龄13年 暂无认证

125	3万+	6万+	4万+	
原创	周排名	总排名	访问	等级
1632	14	70	26	142
积分	粉丝	获赞	评论	收藏

私信 关注

搜博主文章

热门文章

MongoDB和Elasticsearch的各使用场景对比 1659
SpringFox 3尝鲜 集成SpringBoot生成Swagger接口文档 1348
clickhouse设置用户名密码 1281
K8s宣布弃用Docker，千万别慌！ 1201
Rabbitmq如何保证消息顺序执行 1182

分类专栏

elasticsearch 5篇
运维 35篇
面试相关 22篇
开发问题集锦 17篇
mongodb 2篇
mysql 6篇

最新评论

消息中间件—RocketMQ消息存储
Tisfy: 我毫不犹豫的把楼主的这个帖子收藏了
Nacos 安装部署详细说明
tttggg1631: 写的不错，加油。
java.lang.AbstractMethodError: com.alib…
BigBossc: 我也遇见了这个错误，我的版本是Springcloud Hoxton.SR1 和Alibaba 2 …
SpringFox 3尝鲜 集成SpringBoot生成Sw…
中布斯: 很奇怪，又不报错了；
Spring Boot+JWT快速实现简单的接口鉴权
Thexingxin: 不齐全的吗

最新文章

ElasticSearch 实战之千万级 TPS 写入
easyconnect(mac版)总是初始化问题
业界前所未有：10 分钟部署十万量级资源、1 小时完成微博后端异地重建

2021年 32篇 2020年 140篇

SpringBoot 集成Kafka操作详解

转载 时间都哪去了 2020-10-23 10:15:54 711 收藏 7 版权
分类专栏： kafka springboot系列

目录[-]

参考信息：

- kafka 官方网址
- spring-kafka 2.2.7 版本文档
- 示例 Github 地址： https://github.com/my-dlq/blog-example/tree/master/springboot/springboot-kafka

环境说明：

- Kafka 版本：2.3.0
- Zookeeper 版本：3.4.14
- SpringBoot 版本：2.1.7.RELEASE
- Spring For Apache Kafka 版本：2.2.8

一、概念知识

什么是消息中间件

消息中间件利用高效可靠的消息传递机制进行平台无关的数据交流，并基于数据通信来进行分布式系统的集成。通过提供消息传递和消息排队模型，它可以在分布式环境下扩展进程间的通信。

什么是 Kafka

Apache Kafka 是一个分布式高吞吐量的流消息系统，Kafka 建立在 ZooKeeper 同步服务之上。它与 Apache Storm 和 Spark 完美集成，用于实时流数据分析，与其他消息传递系统相比，Kafka具有更好的吞吐量，内置分区，数据副本和高度容错功能，因此非常适合大型消息处理应用场景。

Kafka 特性

- **高并发：** 支持数千个客户端同时读写。
- **可扩展性：** kafka集群支持热扩展。
- **容错性：** 允许集群中节点失败(若副本数量为n，则允许n-1个节点失败)。
- **持久性、可靠性：** 消息被持久化到本地磁盘，并且支持数据备份防止数据丢失。
- **高吞吐量、低延迟：** Kafka每秒可以处理几十万消息，延迟最低只有几毫秒，每个消息主题topic可以分多个区，消费者组(consumer group)对消息分区(partition)进行消费。

使用场景

- **日志收集：** 可以用 kafka 收集各种服务的日志，通过kafka以统一接口服务的方式开放给各种消费者，如 hadoop，Hbase，Solr 等。
- **消息系统：** 解耦生产者和消费者、缓存消息等。
- **用户活动跟踪：** Kafka 经常被用来记录web用户或者app用户的各种活动，如浏览网页，搜索，点击等活动，这些活动信息被各个服务器发布到 kafka 的 topic 中，然后订阅者通过订阅这些 topic 来做实时的监控分析，或者装载到 hadoop、数据仓库中做离线分析和挖掘。
- **运营指标：** Kafka也经常用来记录运营监控数据，包括收集各种分布式应用的数据，比如报警和报告等。
- **流式处理：** 比如 spark streaming 和 storm。

基本概念

- **Broker：** 消息中间件处理节点，一个 Kafka 节点就是一个 Broker，一个或者多个 Broker 可以组成一个 Kafka 集群。
- **Topic：** Kafka 的消息通过 Topic 主题来分类，Topic类似于关系型数据库中的表，每个 Topic 包含一个或多（Partition）分区。
- **Partition：** 多个分区会分布在Kafka集群的不同服务节点上，消息以追加的方式写入一个或多个分区中。
- **LogSegment：** 每个分区又被划分为多个日志分段 LogSegment 组成，日志段是 Kafka 日志对象分片的最小单位。LogSegment 算是一个逻辑概念，对应一个具体的日志文件（".log" 的数据文件）和两个索引文件（".index" 和 ".timeindex"，分别表示偏移量索引文件和消息时间戳索引文件）组成。
- **Offset：** 每个分区中都由一系列有序的、不可变的消息组成，这些消息被顺序地追加到 Partition 中，每个消息都有一个连续的序列号称之为 Offset 偏移量，用于在 Partition 内唯一标识消息。
- **Message：** 消息是 Kafka 中存储的最小最基本的单位，即为一个 commit log，由一个固定长度的消息头和一个可变长度的消息体组成。
- **Producer：** 消息的生产者，负责发布消息到 Kafka Broker，生产者在默认情况下把消息均衡地分布到主题的所有分区上，用户也可以自定义分区器来实现消息的分区路由。
- **Consumer：** 消息的消费者，从 Kafka Broker 读取消息的客户端，消费者把每个分区最后读取的消息的 Offset 偏移量保存在 Zookeeper 或 Kafka 上，如果消费者关闭或重启，它的读取状态不会丢失。
- **Consumer Group：** 每个 Consumer 属于一个特定的 Consumer Group（若不指定 Group Name则属于默认的 group），一个或多个 Consumer 组成的群组可以共同消费一个 Topic 中的消息，但每个分区只能被群组中的一个消费者操作。

生产者 ACKS 机制

ACKS 参数指定了必须要有多少个分区副本接收到消息，生产者才会认为消息写入是发送消息成功的，这个参数对消息丢失的可能性会产生重要影响，主参数有如下选项：

- **acks=0：** 把消息发送到kafka就认为发送成功。
- **acks=1：** 把消息发送到kafka leader分区，并且写入磁盘就认为发送成功。

- **acks=all**：把消息发送到 Kafka Leader 分区，并且 Leader 分区的副本 Follower 对消息进行了同步就认为发送成功。

消费者更新 Offset 偏移量两种方式

> 详情可以查看参考的一篇文章：https://www.jianshu.com/p/d5cd34e429a2

消费者把每个分区最后读取的悄息偏移量提交保存在 Zookeeper 或 Kafka 上，如果消费者关闭或重启，它的读取状态不会丢失，KafkaConsumer API 提供了很多种方式来提交偏移量，但是不同的提交方式会产生不同的数据影响。

- **自动提交**：

如果 `enable.auto.commit` 被设置为 true，那么消费者会自动提交当前处理到的偏移量存入 Zookeeper，自动提交的时间间隔为5s，通过 `atuo.commit.interval.ms` 属性设置，自动提交是非常方便，但是自动提交会出现消息被重复消费的风险，可以通过修改提交时间间隔来更频繁地提交偏移量，减小可能出现重复悄息的时间窗，不过这种情况是无也完全避免的。

- **手动提交**：

鉴于 Kafka 自动提交 Offset 的不灵活性和不精确性（只能是按指定频率的提交），Kafka提供了手动提交 Offset 策略，将 `auto.commit.offset` 自动提交参数设置为 false 来关闭自动提交开启手动模式，手动提交能对偏移量更加灵活精准地控制，以保证消息不被重复消费以及消息不被丢失。

二、SpringBoot 操作 Kafka 示例

1、Maven 引入 Kafka 相关组件

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.or
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.1.7.RELEASE</version>
    </parent>

    <groupId>club.mydlq</groupId>
    <artifactId>springboot-kafka-demo</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>springboot-kafka-demo</name>

    <properties>
        <java.version>1.8</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.kafka</groupId>
            <artifactId>spring-kafka</artifactId>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

2、Topic 配置

配置 Topic，每次程序启动时检测 Kafka 中是否存在已经配置的 Topic，如果不存在就创建。

```
import org.apache.kafka.clients.admin.AdminClientConfig;
import org.apache.kafka.clients.admin.NewTopic;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.core.KafkaAdmin;
import java.util.HashMap;
import java.util.Map;

@Configuration
public class KafkaTopicConfig {

    /**
     * 定义一个KafkaAdmin的bean，可以自动检测集群中是否存在topic，不存在则创建
     */
    @Bean
    public KafkaAdmin kafkaAdmin() {
        Map<String, Object> configs = new HashMap<>();

        // 指定多个kafka集群多个地址，例如：192.168.2.11,9092,192.168.2.12:9092,192.168.2.13
        configs.put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG,"127.0.0.1:9092");
        return new KafkaAdmin(configs);
    }

    /**
     * 创建 Topic
     */
    @Bean
    public NewTopic topicinfo() {
        // 创建topic，需要指定创建的topic的"名称"、"分区数"、"副本数量(副本数数目设置要小于Broker数
        return new NewTopic("test", 3, (short) 0);
    }

}
```

3、Producer 配置

(1)、创建 Producer 配置类

创建 Producer 配置类，对 Kafka 生产者进行配置,在配置中需要设置三个 Bean 分别为：

- kafkaTemplate：kafka template 实例，用于 Spring 中的其它对象引入该 Bean，通过其向 Kafka 发送消息。
- producerFactory：producer 工厂，用于对 kafka producer 进行配置。
- producerConfigs：对 kafka producer 参数进行配置。

```
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.common.serialization.StringSerializer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.annotation.EnableKafka;
import org.springframework.kafka.core.DefaultKafkaProducerFactory;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.kafka.core.ProducerFactory;
import java.util.HashMap;
import java.util.Map;

// 设置@Configuration、@EnableKafka两个注解，声明Config并且打开KafkaTemplate能力。
```

```
public class KafkaProducerConfig {
                                    16
    /**
     * Producer Template 配置
     */
    @Bean(name="kafkaTemplate")
    public KafkaTemplate<String, String> kafkaTemplate() {
        return new KafkaTemplate<>(producerFactory());
    }

    /**
     * Producer 工厂配置
     */
    public ProducerFactory<String, String> producerFactory() {
        return new DefaultKafkaProducerFactory<>(producerConfigs());
    }

    /**
     * Producer 参数配置
     */
    public Map<String, Object> producerConfigs() {
        Map<String, Object> props = new HashMap<>();
        // 指定多个kafka集群多个地址
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,"127.0.0.1:9092,127.0.0.1:9
        // 重试次数，0为不启用重试机制
        props.put(ProducerConfig.RETRIES_CONFIG, 0);
        // acks=0 把消息发送到kafka就认为发送成功
        // acks=1 把消息发送到kafka leader分区，并且写入磁盘就认为发送成功
        // acks=all 把消息发送到kafka leader分区，并且leader分区的副本follower对消息进行了同步
        props.put(ProducerConfig.ACKS_CONFIG,"1");
        // 生产者空间不足时，send()被阻塞的时间，默认60s
        props.put(ProducerConfig.MAX_BLOCK_MS_CONFIG, 6000);
        // 控制批处理大小，单位为字节
        props.put(ProducerConfig.BATCH_SIZE_CONFIG, 4096);
        // 批量发送，延迟为1毫秒，启用该功能能有效减少生产者发送消息次数，从而提高并发量
        props.put(ProducerConfig.LINGER_MS_CONFIG, 1);
        // 生产者可以使用的总内存字节来缓冲等待发送到服务器的记录
        props.put(ProducerConfig.BUFFER_MEMORY_CONFIG, 40960);
        // 消息的最大大小限制,也就是说send的消息大小不能超过这个限制, 默认1048576(1MB)
        props.put(ProducerConfig.MAX_REQUEST_SIZE_CONFIG,1048576);
        // 键的序列化方式
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class)
        // 值的序列化方式
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.clas
        // 压缩消息，支持四种类型，分别为：none、lz4、gzip、snappy，默认为none。
        // 消费者默认支持解压，所以压缩设置在生产者，消费者无需设置。
        props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG,"none");
        return props;
    }

}
```

(2)、创建 Producer Service 向 kafka 发送数据

创建 Producer Service 引入 KafkaTemplate 对象，再创建 sendMessageSync、sendMessageAsync 两个方法，分别利用“同步/异步”两种方法向 kafka 发送消息。

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.kafka.support.SendResult;
import org.springframework.stereotype.Service;
import org.springframework.util.concurrent.ListenableFuture;
import org.springframework.util.concurrent.ListenableFutureCallback;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

@Service
public class KafkaProducerService {

    @Autowired
    private KafkaTemplate kafkaTemplate;

    /**
     * producer 同步方式发送数据
     * @param topic    topic名称
     * @param message  producer发送的数据
     */
    public void sendMessageSync(String topic, String message) throws InterruptedExcep
        kafkaTemplate.send(topic, message).get(10, TimeUnit.SECONDS);
    }

    /**
     * producer 异步方式发送数据
     * @param topic    topic名称
     * @param message  producer发送的数据
     */
    public void sendMessageAsync(String topic, String message) {
        ListenableFuture<SendResult<Integer, String>> future = kafkaTemplate.send(top
        future.addCallback(new ListenableFutureCallback<SendResult<Integer, String>>(
            @Override
            public void onSuccess(SendResult<Integer, String> result) {
                System.out.println("success");
            }

            @Override
            public void onFailure(Throwable ex) {
                System.out.println("failure");
            }
        });
    }

}
```

(3)、创建 Producer Controller 调用 Producer Service 产生数据

Spring Controller 类，用于调用 Producer Service 中的方法向 kafka 发送消息。

```
import club.mydlq.springbootkafkademo.service.ProducerService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.TimeoutException;

@RestController
public class KafkaProducerController {

    @Autowired
    private KafkaProducerService producerService;

    @GetMapping("/sync")

 public void sendMessageSync() throws InterruptedException, ExecutionException, Time
        producerService.sendMessageSync("test","同步发送消息测试");17     }

    @GetMapping("/async")
    public void sendMessageAsync(){
        producerService.sendMessageAsync("test","异步发送消息测试");
    }

}
```

4、Consumer 配置

(1)、创建 Consumer 配置类

创建 Consumer 配置类，对 Kafka 消费者进行配置,在配置中需要设置三个 Bean 分别为：

- kafkaListenerContainerFactory：kafka container 工厂，负责创 建container，当使用@KafkaListener时需要提供。
- consumerFactory：consumer 工厂，用于对 kafka consumer 进行配置。
- consumerConfigs：对 kafka consumer 参数进行配置。

```
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.common.serialization.StringDeserializer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.annotation.EnableKafka;
import org.springframework.kafka.config.ConcurrentKafkaListenerContainerFactory;
import org.springframework.kafka.config.KafkaListenerContainerFactory;
import org.springframework.kafka.core.ConsumerFactory;
import org.springframework.kafka.core.DefaultKafkaConsumerFactory;
import org.springframework.kafka.listener.ConcurrentMessageListenerContainer;
import java.util.HashMap;
import java.util.Map;

@Configuration
@EnableKafka
public class KafkaConsumerConfig {

    @Bean
    KafkaListenerContainerFactory<ConcurrentMessageListenerContainer<Integer, String>
        ConcurrentKafkaListenerContainerFactory<Integer, String>
                factory = new ConcurrentKafkaListenerContainerFactory<>();
        // 设置消费者工厂
        factory.setConsumerFactory(consumerFactory());
        // 消费者组中线程数量
        factory.setConcurrency(3);
        // 拉取超时时间
        factory.getContainerProperties().setPollTimeout(3000);
        return factory;
    }

    @Bean
    public ConsumerFactory<Integer, String> consumerFactory() {
        return new DefaultKafkaConsumerFactory<>(consumerConfigs());
    }

    @Bean
    public Map<String, Object> consumerConfigs() {
        Map<String, Object> propsMap = new HashMap<>();
        // Kafka地址
        propsMap.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "127.0.0.1:9092,127.0.0
        // 是否自动提交offset偏移量(默认true)
        propsMap.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, true);
        // 自动提交的频率(ms)
        propsMap.put(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG, "100");
        // Session超时设置
        propsMap.put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, "15000");
        // 键的反序列化方式
        propsMap.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer
        // 值的反序列化方式
        propsMap.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializ
        // offset偏移量规则设置：
        // (1)、earliest：当各分区下有已提交的offset时，从提交的offset开始消费；无提交的offset
        // (2)、latest：当各分区下有已提交的offset时，从提交的offset开始消费；无提交的offset时，
        // (3)、none：topic各分区都存在已提交的offset时，从offset后开始消费；只要有一个分区不存
        propsMap.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
        return propsMap;
    }

}
```

(2)、创建 Consumer Service 监听 Kafka 数据

```
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.stereotype.Service;

@Service
public class KafkaConsumerService {

    @KafkaListener(topics = {"test"},groupId = "group1", containerFactory="kafkaListe
    public void kafkaListener(String message){
        System.out.println(message);
    }

}
```

三、SpringBoot 操作 Kafka 详解

1、Producer Template 发送消息几种方法

KafkaTemplate 类提供了非常方便的方法将数据发送到 kafka 的 Topic，以下清单显示了该类的提供的相关方法，详情可以查看 KafkaTemplate 类方法文档

```
// 设定data，向kafka发送消息
ListenableFuture<SendResult<K, V>> sendDefault(V data);
// 设定key、data，向kafka发送消息
ListenableFuture<SendResult<K, V>> sendDefault(K key, V data);
// 设定partition、key、data，向kafka发送消息
ListenableFuture<SendResult<K, V>> sendDefault(Integer partition, K key, V data);
// 设定partition、timestamp、key、data，向kafka发送消息
ListenableFuture<SendResult<K, V>> sendDefault(Integer partition, Long timestamp, K k
// 设定topic、data，向kafka发送消息
ListenableFuture<SendResult<K, V>> send(String topic, V data);
// 设定topic、key、data，向kafka发送消息
ListenableFuture<SendResult<K, V>> send(String topic, K key, V data);
// 设定topic、partition、key、data，向kafka发送消息
ListenableFuture<SendResult<K, V>> send(String topic, Integer partition, K key, V dat
// 设定topic、partition、timestamp、 key、data，向kafka发送消息
ListenableFuture<SendResult<K, V>> send(String topic, Integer partition, Long timesta
// 创建ProducerRecord对象，在ProducerRecord中设置好topic、partion、key、value等信息，然后向k
ListenableFuture<SendResult<K, V>> send(ProducerRecord<K, V> record);
// 创建Spring的Message对象，然后向kafka发送消息
ListenableFuture<SendResult<K, V>> send(Message<?> message);
// 获取指标信息
Map<MetricName, ? extends Metric> metrics();
// 显示Topic分区信息
List<PartitionInfo> partitionsFor(String topic);
//在生产者上执行一些任意操作并返回结果。
<T> T execute(ProducerCallback<K, V, T> callback);
// 生产者刷新消息
void flush();

// 用于执行生产者方法后异步回调
interface ProducerCallback<K, V, T> {
    T doInKafka(Producer<K, V> producer);
}
```

下面将写个使用示例，这里改下上面向 kafka service 发送数据的例子，通过不同的方法向 kafka 发送消息，具体代码如下：

```
import org.apache.kafka.clients.producer.ProducerRecord;
import org.springframework.beans.factory.annotation.Autowired;
```

```
import org.springframework.kafka.support.KafkaHeaders;
import org.springframework.messaging.Message;
import org.springframework.messaging.support.MessageBuilder;
import org.springframework.stereotype.Service;
import org.springframework.util.concurrent.ListenableFuture;
import org.springframework.util.concurrent.ListenableFutureCallback;
import java.util.Date;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

@Service
public class ProducerService {

    @Autowired
    private KafkaTemplate kafkaTemplate;

    /**
     * producer 同步方式发送数据
     *
     * @param topic   topic名称
     * @param message producer发送的数据
     */
    public void sendMessageSync(String topic, String key, String message) throws Int
        //------- 方法：send(String topic, @Nullable V data)
        kafkaTemplate.send(topic, message).get(10, TimeUnit.SECONDS);
        //------- 方法：send(String topic, K key, @Nullable V data)
        kafkaTemplate.send(topic, key, message).get(10, TimeUnit.SECONDS);
        //------- 方法：send(String topic, K key, @Nullable V data)
        kafkaTemplate.send(topic, 0, message).get(10, TimeUnit.SECONDS);
        //------- 方法：send(String topic, Integer partition, K key, @Nullable V data,
        kafkaTemplate.send(topic, 0, key, message).get(10, TimeUnit.SECONDS);
        //------- 方法：send(String topic, Integer partition, Long timestamp, K key, (
        kafkaTemplate.send(topic, 0, new Date().getTime(),key, message).get(10, Timel
        //------- 方法：send(Message<?> message)
        Message msg = MessageBuilder.withPayload("Send Message(payload,headers) Test
                .setHeader(KafkaHeaders.MESSAGE_KEY, key)
                .setHeader(KafkaHeaders.TOPIC, topic)
                .setHeader(KafkaHeaders.PREFIX,"kafka_")
                .build();
        kafkaTemplate.send(msg).get(10, TimeUnit.SECONDS);
        //------- 方法：send(ProducerRecord<K, V> record)
        ProducerRecord<String, String> producerRecord1 = new ProducerRecord<>("test"
        ProducerRecord<String, String> producerRecord2 = new ProducerRecord<>("test"
        kafkaTemplate.send(producerRecord1).get(10, TimeUnit.SECONDS);
        kafkaTemplate.send(producerRecord2).get(10, TimeUnit.SECONDS);
    }

    /**
     * producer 异步方式发送数据
     *
     * @param topic   topic名称
     * @param message producer发送的数据
     */
    public void sendMessageAsync(String topic, String key, String message) {
        //------- 方法：send(String topic, @Nullable V data)
        ListenableFuture<SendResult<Integer, String>> future1 = kafkaTemplate.send(t
        //------- 方法：send(String topic, K key, @Nullable V data)
        ListenableFuture<SendResult<Integer, String>> future2 = kafkaTemplate.send(t
        //------- 方法：send(String topic, K key, @Nullable V data)
        ListenableFuture<SendResult<Integer, String>> future3 = kafkaTemplate.send(t
        //------- 方法：send(String topic, Integer partition, K key, @Nullable V data,
        ListenableFuture<SendResult<Integer, String>> future4 = kafkaTemplate.send(t
        //------- 方法：send(String topic, Integer partition, Long timestamp, K key, (
        ListenableFuture<SendResult<Integer, String>> future5 = kafkaTemplate.send(t
        //------- 方法：send(Message<?> message)
                Message msg = MessageBuilder.withPayload("Send Message(payload,heade
                .setHeader(KafkaHeaders.MESSAGE_KEY, key)
                .setHeader(KafkaHeaders.TOPIC, topic)
                .setHeader(KafkaHeaders.PREFIX,"kafka_")
                .build();
        ListenableFuture<SendResult<Integer, String>> future6 = kafkaTemplate.send(m
        //------- 方法：send(ProducerRecord<K, V> record)
        ProducerRecord<String, String> producerRecord1 = new ProducerRecord<>("test"
        ProducerRecord<String, String> producerRecord2 = new ProducerRecord<>("test"
        ListenableFuture<SendResult<Integer, String>> future7 = kafkaTemplate.send(p
        ListenableFuture<SendResult<Integer, String>> future8 = kafkaTemplate.send(p

        // 设置异步发送消息获取发送结果后执行的动作
        ListenableFutureCallback listenableFutureCallback = new ListenableFutureCall
            @Override
            public void onSuccess(SendResult<Integer, String> result) {
                System.out.println("success");
            }

            @Override
            public void onFailure(Throwable ex) {
                System.out.println("failure");
            }
        };
        // 将listenableFutureCallback与异步发送消息对象绑定
        future1.addCallback(listenableFutureCallback);
        future2.addCallback(listenableFutureCallback);
        future3.addCallback(listenableFutureCallback);
        future4.addCallback(listenableFutureCallback);
        future5.addCallback(listenableFutureCallback);
        future6.addCallback(listenableFutureCallback);
        future7.addCallback(listenableFutureCallback);
        future8.addCallback(listenableFutureCallback);
    }

}
```

2、Kafka Consumer 监听 Kafka 消息

当我们需要接收 kafka 中的消息时需要使用消息监听器，Spring For Kafka 提供了八种消息监听器接口，接口如下：

```
/**
 * 当使用"自动提交"或"ontainer-managed"中一个提交方法提交offset偏移量时，
 * 使用此接口处理Kafka consumer poll()操作接收到的各个ConsumerRecord实例。
 */
public interface MessageListener<K, V> {
    void onMessage(ConsumerRecord<K, V> data);
}
/**
 * 当使用手动提交offset偏移量时，使用此接口处理从Kafka consumer poll()操作接收到的各个ConsumerR
 */
public interface AcknowledgingMessageListener<K, V> {
    void onMessage(ConsumerRecord<K, V> data, Acknowledgment acknowledgment);
}
/**
 * 当使用"自动提交"或"ontainer-managed"中一个提交方法提交offset偏移量时，
 * 使用此接口处理Kafka consumer poll()操作接收到的各个ConsumerRecord
 * 实例。并提供可访问的consumer对象。
 */
public interface ConsumerAwareMessageListener<K, V> extends MessageListener<K, V> {
    void onMessage(ConsumerRecord<K, V> data, Consumer<?, ?> consumer);
}
/**
 * 当使用手动提交offset偏移量时，使用此接口处理从Kafka consumer poll()操作
 * 接收到的各个ConsumerRecord实例。并提供可访问的consumer对象。
 */
public interface AcknowledgingConsumerAwareMessageListener<K, V> extends MessageListe
    void onMessage(ConsumerRecord<K, V> data, Acknowledgment acknowledgment, Consumer
}
```

```
 * 使用此接口处理从Kafka consumer poll()操作接收到的所有ConsumerRecord实例。
 *
 * 注意：使用此接口时不支持ACK的AckMode.RECORD模式，因为监听器已获得完整的批处理。
 */
public interface BatchMessageListener<K, V> {
    void onMessage(List<ConsumerRecord<K, V>> data);
}
/**
 * 当使用手动提交offset偏移量时，使用此接口处理从Kafka consumer poll()操作接收到的所有ConsumerR
 */
public interface BatchAcknowledgingMessageListener<K, V> {
    void onMessage(List<ConsumerRecord<K, V>> data, Acknowledgment acknowledgment);
}
/**
 * 当使用"自动提交"或"ontainer-managed"中一个提交方法提交offset偏移量时，
 * 使用此接口处理从Kafka consumer poll()操作接收到的所有ConsumerRecord实例。
 * 并提供可访问的consumer对象。
 *
 * 注意：使用此接口时不支持ACK的AckMode.RECORD模式，因为监听器已获得完整的批处理。
 */
public interface BatchConsumerAwareMessageListener<K, V> extends BatchMessageListener
    void onMessage(List<ConsumerRecord<K, V>> data, Consumer<?, ?> consumer);
}
/**
 * 当使用手动提交offset偏移量时，使用此接口处理从Kafka consumer poll()操作接收到的
 * 所有ConsumerRecord实例。并提供可访问的consumer对象。
 */
public interface BatchAcknowledgingConsumerAwareMessageListener<K, V> extends BatchMe
    void onMessage(List<ConsumerRecord<K, V>> data, Acknowledgment acknowledgment, Co
}
```

上面接口中的方法归总就是：

序号	消费方式	自动提交Offset偏移量	提供Consumer对象
1	单条	是	否
2	单条	否	否
3	单条	是	是
4	单条	否	是
5	批量	是	否
6	批量	否	否
7	批量	是	是
8	批量	否	是

Spring For Kafka 提供了消息监听器接口的两种实现类，分别是：

- KafkaMessageListenerContainer
- ConcurrentMessageListenerContainer

KafkaMessageListenerContainer 利用单个线程来接收全部主题中全部分区上的所有消息。
ConcurrentMessageListenerContainer 代理的一个或多个 KafkaMessageListenerContainer 实例，来实现多个线程消费。

下面将创建一个 KafkaMessageListenerContainer 实例来监听 Kafka 消息：

```
@Configuration
@EnableKafka
public class ConsumerConfigDemo {
    @Bean
    public Map<String, Object> consumerConfigs() {
        Map<String, Object> propsMap = new HashMap<>();
        propsMap.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "127.0.0.1:9092");
        propsMap.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, true);
        propsMap.put(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG, "100");
        propsMap.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.c
        propsMap.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer
        return propsMap;
    }
    @Bean
    public ConsumerFactory<Integer, String> consumerFactory() {
        return new DefaultKafkaConsumerFactory<>(consumerConfigs());
    }

    /**
     * 创建 KafkaMessageListenerContainer 实例监听 kafka 消息
     */
    @Bean
    public KafkaMessageListenerContainer demoListenerContainer() {
        // 创建container配置参数，并指定要监听的 topic 名称
        ContainerProperties properties = new ContainerProperties("test");
        // 设置消费者组名称
        properties.setGroupId("group2");
        // 设置监听器监听 kafka 消息
        properties.setMessageListener(new MessageListener<Integer,String>() {
            @Override
            public void onMessage(ConsumerRecord<Integer, String> record) {
                System.out.println("消息: " + record);
            }
        });
        return new KafkaMessageListenerContainer(consumerFactory(), properties);
    }

}
```

上面示例启动后将监听 topic 名称为 “test” 的 kafka 消息，不过这样启动只是单线程消费，如果想多线程消费就得创建多个实例来监控该 topic 不同的分区。但是这样操作来完成消费者多线程消费比较麻烦，所以一般使用 Spring For Kafka 组件时都会创建 `KafkaListenerContainerFactory` Bean 来代理多个 `KafkaMessageListenerContainer` 完成消费者多线程消费。

3、使用 @KafkaListener 注解监听 Kafka 消息

为了使创建 kafka 监听器更加简单，Spring For Kafka 提供了 `@KafkaListener` 注解，该 `@KafkaListener` 注解配置方法上，凡是带上此注解的方法就会被标记为是 Kafka 消息监听器，所以可以用 `@KafkaListener` 注解快速创建消息监听器。

下面写几个例子来简单描述下使用方法:

(1)、监听单个 Topic 示例

这里先写一个简单使用 @KafkaListener 完成消息监听的示例。

```
@Configuration
@EnableKafka
public class ConsumerConfigDemo {
    @Bean
    public Map<String, Object> consumerConfigs() {
        Map<String, Object> propsMap = new HashMap<>();
        propsMap.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "127.0.0.1:9092");
        propsMap.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, true);
        propsMap.put(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG, "100");
        propsMap.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer
        propsMap.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializ
        return propsMap;
    }

    @Bean
    public ConsumerFactory<Integer, String> consumerFactory() {
```

```
    }19
    @Bean
    KafkaListenerContainerFactory<ConcurrentMessageListenerContainer<Integer, String>
        ConcurrentKafkaListenerContainerFactory<Integer, String>
                factory = new ConcurrentKafkaListenerContainerFactory<>();
        factory.setConsumerFactory(consumerFactory());
        // 创建3个线程并发消费
        factory.setConcurrency(3);
        // 设置拉取数据超时时间
        factory.getContainerProperties().setPollTimeout(3000);
        return factory;
    }

    /**
     *  ---使用@KafkaListener注解来标记此方法为kafka消息监听器，创建消费组group1监听test topi
     */
    @KafkaListener(topics = {"test"},groupId = "group1")
    public void kafkaListener(String message){
        System.out.println("消息："+message);
    }

}
```

(2)、监听多个 Topic 示例

使用 @KafkaListener 也可以监控多个 topic 的消息，示例如下：

```
@KafkaListener(topics = {"test1", "test2"}, groupId = "group1")
public void kafkaListener(String message){
    System.out.println("消息："+message);
}
```

(3)、监听某个 Topic 的某个分区示例

单独监听某个分区息，示例如下：

```
@KafkaListener(id = "id0", groupId = "group1", topicPartitions = { @TopicPartition(to
public void kafkaListener1(String message) {
    System.out.println("消息："+message);
}

@KafkaListener(id = "id1", groupId = "group1", topicPartitions = { @TopicPartition(to
public void kafkaListener2(String message) {
    System.out.println("消息："+message);
}
```

(4)、监听多个 Topic 的分区示例

同时监听多个 topic 的分区，示例如下：

```
@KafkaListener(id = "test", group = "group1", topicPartitions = {
            @TopicPartition(topic = "test1", partitions = {"0"}),
            @TopicPartition(topic = "test2", partitions = {"0", "1"})
        })
public void kafkaListener(String message) {
    System.out.print(message);
}
```

(5)、获取监听的 topic 消息头中的元数据

可以从消息头中获取有关消息的元数据，例如：

```
@KafkaListener(topics = "test", groupId = "group1")
public void kafkaListener(@Payload String message, @Header(KafkaHeaders.RECEIVED_TOPI
                                                   @Header(KafkaHeaders.RECEIVED_MESSA
    System.out.println("主题:" + topic);
    System.out.println("键key:" + key);
    System.out.println("消息:" + message);
}
```

(6)、监听 topic 进行批量消费

如果参数配置中设置为批量消费，则 @KafkaListener 注解的方法的参数要使用 List 来接收，例如：

```
@KafkaListener(topics = "test", groupId = "group1")
public void kafkaListener(List<String> messages) {
    for(String msg:messages){
        System.out.println(msg);
    }
}
```

(7)、监听 topic 并手动提交 Offset 偏移量

如果设置为手动提交 Offset 偏移量，并且设置 Ack 模式为 MANUAL 或 MANUAL_IMMEDIATE,则需要在方法参数中引入 Acknowledgment 对象，并执行它的 acknowledge() 方法来提交偏移量。

```
@KafkaListener(topics = "test",groupId = "group5")
public void kafkaListener(List<String> messages, Acknowledgment acknowledgment) {
    for(String msg:messages){
        System.out.println(msg);
    }
    // 触发提交offset偏移量
    acknowledgment.acknowledge();
}
```

4、使用 @KafkaListener 模糊匹配多个 Topic

使用 @KafkaListener 注解时，可以添加参数 topicPattern ，输入通配符来对多个 topic 进行监听，例如这里使用 “test.*” 将监听所有以 test 开头的 topic 的消息。

```
@KafkaListener(topicPattern = "test.*",groupId = "group6")
public void annoListener2(String messages) {
    System.err.println(messages);
}
```

5、使用 @SendTo 注解转发消息

在平时处理业务逻辑时候，经常需要接收 kafka 中某个 topic 的消息，进行一系列处理来完成业务逻辑，然后再进行转发到一个新的 topic 中，由于这种业务需求，Spring For Kafka 提供了 @SendTo 注解，只要在 @KafkaListener 与 @SendTo 注解在同一个方法上，并且该方法存在返回值，那么就能将监听的数据在方法内进行处理后 return，然后转发到 @SendTo 注解内设置的 topic 中。

完成上面操作需要几个步骤：

1. 配置 Producer 参数，并创建 kafkaTemplate Bean。
2. 配置KafkaListenerContainerFactory的ReplyTemplate，将 kafkaTemplate 对象添加到其中。
3. 创建消息监听器方法，设置该方法拥有返回值，并添加 @KafkaListener 与 @SendTo 两个注解，并在 @SendTo 注解中输入消息转发的 Topic。

(1)、配置 Producer 参数，并创建 kafkaTemplate Bean

```
@Configuration
@EnableKafka
```

```
    /**
     * kafkaTemplate Bean
     */
    @Bean(name="kafkaTemplate")
    public KafkaTemplate<String, String> kafkaTemplate() {
        return new KafkaTemplate<>(producerFactory());
    }

    public ProducerFactory<String, String> producerFactory() {
        return new DefaultKafkaProducerFactory<>(producerConfigs());
    }

    public Map<String, Object> producerConfigs() {
        Map<String, Object> props = new HashMap<>();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,"127.0.0.1:9092");

        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);

        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class)
        return props;
    }
}
```

(2)、配置KafkaListenerContainerFactory的ReplyTemplate，将 kafkaTemplate 对象添加到其中

```
@Configuration
@EnableKafka
public class KafkaConsumerConfig {

    @Autowired
    private KafkaTemplate kafkaTemplate;

    @Bean
    KafkaListenerContainerFactory<ConcurrentMessageListenerContainer<Integer, String>
        ConcurrentKafkaListenerContainerFactory<Integer, String> factory = new Concur
        factory.setConsumerFactory(consumerFactory());
        factory.setConcurrency(3);
        factory.getContainerProperties().setPollTimeout(3000);
        // ---设置ReplyTemplate参数，将kafkaTemplate对象加入
        factory.setReplyTemplate(kafkaTemplate);
        return factory;
    }

    @Bean
    public ConsumerFactory<Integer, String> consumerFactory() {
        return new DefaultKafkaConsumerFactory<>(consumerConfigs());
    }

    @Bean
    public Map<String, Object> consumerConfigs() {
        Map<String, Object> propsMap = new HashMap<>();
        propsMap.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "127.0.0.1:9092");
        propsMap.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, true);
        propsMap.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer
        propsMap.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializ
        return propsMap;
    }
```

(3)、创建消息监听器方法，设置该方法拥有返回值，并添加 `@KafkaListener` 与 `@SendTo` 两个注解，并在 @SendTo 注解中输入消息转发的 Topic。

```
@Service
public class KafkaConsumerMessage {

    /**
     * 监听test1 topic,设置返回值为string类型，并添加@SendTo注解，将消息转发到 test2
     */
    @KafkaListener(topics = "test1",groupId = "group1")
    @SendTo("test2")
    public String kafkaListener1(String messages) {
        System.out.println(messages);
        String newMsg = messages + "消息转发测试";
        // 将处理后的消息返回
        return newMsg;
    }

    /**
     * 监听 test2 topic
     */
    @KafkaListener(topics = "test2",groupId = "group2")
    public void kafkaListener2(String messages) {
        System.err.println(messages);
    }
}
```

6、Kafka Consumer 并发批量消费消息

(1)、设置并发数与开启批量

- kafkaListenerContainerFactory 设置 `factory.setConcurrency(3)` 设置并发，这个值不能超过topic分区数目
- kafkaListenerContainerFactory 设置 `factory.setBatchListener(true)` 开启批量
- consumerConfigs 配置 `ConsumerConfig.MAX_POLL_RECORDS_CONFIG` 值，来设置批量消费每次最多消费多少条消息记录

```
@Configuration
@EnableKafka
public class ConsumerConfigDemo1 {

    @Bean
    KafkaListenerContainerFactory<ConcurrentMessageListenerContainer<Integer, String>
        ConcurrentKafkaListenerContainerFactory<Integer, String>
                factory = new ConcurrentKafkaListenerContainerFactory<>();
        factory.setConsumerFactory(consumerFactory());
        // 消费者组中线程数量，例如topic有3个分区，为了加快消费将并发设置为3
        factory.setConcurrency(3);
        // 拉取超时时间
        factory.getContainerProperties().setPollTimeout(3000);
        // 当使用批量监听器时需要设置为true
        factory.setBatchListener(true);
        return factory;
    }

    @Bean
    public ConsumerFactory<Integer, String> consumerFactory() {
        return new DefaultKafkaConsumerFactory<>(consumerConfigs());
    }

    @Bean
    public Map<String, Object> consumerConfigs() {
        Map<String, Object> propsMap = new HashMap<>();
        // Kafka地址
        propsMap.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "127.0.0.1:9092");
        // 是否自动提交
        propsMap.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, true);
        // 自动提交的频率
        propsMap.put(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG, "100");
        // Session超时设置
        propsMap.put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, "15000");
        // 键的反序列化方式
        propsMap.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer
        // 值的反序列化方式
        propsMap.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializ
```

```
        return propsMap;
    }

}
```

(2)、设置分区消费

有多个分区的 Topic，可以设置多个注解单独监听 Topic 各个分区以提高效率。

```
@Component
public class ConsumerMessage {

    @KafkaListener(id = "id0", topicPartitions = { @TopicPartition(topic = "test2", p
    public void listenPartition0(List<ConsumerRecord<?, ?>> records) {
        System.out.println("Id0 Listener, Thread ID: " + Thread.currentThread().getId
        System.out.println("Id0 records size " +  records.size());
        for (ConsumerRecord<?, ?> record : records) {
            Optional<?> kafkaMessage = Optional.ofNullable(record.value());
            System.out.println("Received: " + record);
            if (kafkaMessage.isPresent()) {
                Object message = record.value();
                String topic = record.topic();
                System.out.printf(topic + " p0 Received message=" + message);
            }
        }
    }

    @KafkaListener(id = "id1", topicPartitions = { @TopicPartition(topic = "test2", p
    public void listenPartition1(List<ConsumerRecord<?, ?>> records) {
        System.out.println("Id1 Listener, Thread ID: " + Thread.currentThread().getId
        System.out.println("Id1 records size " +  records.size());

        for (ConsumerRecord<?, ?> record : records) {
            Optional<?> kafkaMessage = Optional.ofNullable(record.value());
            System.out.println("Received: " + record);
            if (kafkaMessage.isPresent()) {
                Object message = record.value();
                String topic = record.topic();
                System.out.printf(topic + " p1 Received message=" + message);
            }
        }
    }

    @KafkaListener(id = "id2", topicPartitions = { @TopicPartition(topic = "test2", p
    public void listenPartition2(List<ConsumerRecord<?, ?>> records) {
        System.out.println("Id2 Listener, Thread ID: " + Thread.currentThread().getId
        System.out.println("Id2 records size " +  records.size());

        for (ConsumerRecord<?, ?> record : records) {
            Optional<?> kafkaMessage = Optional.ofNullable(record.value());
            System.out.println("Received: " + record);
            if (kafkaMessage.isPresent()) {
                Object message = record.value();
                String topic = record.topic();
                System.out.printf(topic + " p2 Received message=" + message);
            }
        }
    }

}
```

7、暂停和恢复 Listener Containers

Spring For Kafka 提供 `start()`、`pause()` 和 `resume()` 方法来操作监听容器的启动、暂停和恢复。

- start()：启动监听容器。
- pause()：暂停监听容器。
- resume()：恢复监听容器。

这些方法一般可以灵活操作 kafka 的消费,例如进行服务进行升级，暂停消费者进行消费;例如在白天高峰期不进行服务消费，等到晚上再进行，这时候可以设置定时任务，白天关闭消费者消费到晚上开启;考虑到这些情况，利用 start()、pause()、resume() 这些方法能很好控制消费者进行消费。这里写一个简单例子，通过 cotroller 操作暂停、恢复消费者监听容器。

```
@RestController
public class KafkaController {

    @Autowired
    private KafkaListenerEndpointRegistry registry;

    /**
     * 暂停监听容器
     */
    @GetMapping("/pause")
    public void pause(){
        registry.getListenerContainer("pause.resume").pause();
    }

    /**
     * 恢复监听容器
     */
    @GetMapping("/resume")
    public void resume(){
        //判断监听容器是否启动，未启动则将其启动，否则进行恢复监听容器
        if (!registry.getListenerContainer("pause.resume").isRunning()) {
            registry.getListenerContainer("pause.resume").start();
        }
        registry.getListenerContainer("pause.resume").resume();
    }

}
```

在上面例子中，调用 /pause 接口可以暂停消费者监听容器，调用 /resume 接口可以恢复消费者监听容器。

8、过滤监听器中的消息

在接收消息时候可以创建一个过滤器来过滤接收的消息，这样方便我们不必处理全部消息，只接收我们需要的消息进行处理。

在 kafkaListenerContainerFactory 中配置一个过滤器 `RecordFilterStrategy` 对象过滤消息，这里演示下如何操作：

```
@Configuration
@EnableKafka
public class ConsumerConfigDemo {
    @Bean
    public Map<String, Object> consumerConfigs() {
        Map<String, Object> propsMap = new HashMap<>();
        propsMap.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "127.0.0.1:9092");
        propsMap.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, true);
        propsMap.put(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG, "100");
        propsMap.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer
        propsMap.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializ
        return propsMap;
    }

    @Bean
    public ConsumerFactory<Integer, String> consumerFactory() {
        return new DefaultKafkaConsumerFactory<>(consumerConfigs());
    }

```

```
        ConcurrentKafkaListenerContainerFactory<Integer, String>
                factory = new ConcurrentKafkaListenerContainerFactory<>();
        factory.setConsumerFactory(consumerFactory());
        factory.getContainerProperties().setPollTimeout(3000);
        // 设置过滤器，只接收消息内容中包含 "test" 的消息
        RecordFilterStrategy recordFilterStrategy = new RecordFilterStrategy() {
            @Override
            public boolean filter(ConsumerRecord consumerRecord) {
                String value = consumerRecord.value().toString();
                if (value !=null && value.contains("test")) {
                    System.err.println(consumerRecord.value());
                     // 返回 false 则接收消息
                    return false;
                }
                // 返回 true 则抛弃消息
                return true;
            }
        };
        // 将过滤器添添加到参数中
        factory.setRecordFilterStrategy(recordFilterStrategy);
        return factory;
    }

    /**
     *  监听消息，接收过滤器过滤后的消息
     */
    @KafkaListener(topics = {"test"},groupId = "group1")
    public void kafkaListener(String message){
        System.out.println("消息: "+message);
    }

}
```

9、监听器异常处理

(1)、单消息消费异常处理器

```
@Service
public class ConsumerService {

    /**
     * 消息监听器
     */
    @KafkaListener( topics = {"test"},groupId = "group1",errorHandler = "listenErrorH
    public void listen(String message) {
        System.out.println(message);
        // 创建异常，触发异常处理器
        throw new NullPointerException("测试错误处理器");
    }

    /**
     * 异常处理器
     */
    @Bean
    public ConsumerAwareListenerErrorHandler listenErrorHandler() {
        return new ConsumerAwareListenerErrorHandler() {

            @Override
            public Object handleError(Message<?> message,
                                      ListenerExecutionFailedException e,
                                      Consumer<?, ?> consumer) {
                System.out.println("message:" + message.getPayload());
                System.out.println("exception:" + e.getMessage());
                consumer.seek(new TopicPartition(message.getHeaders().get(KafkaHeader
                              message.getHeaders().get(KafkaHeaders.RECEIVED_PARTIT
                              message.getHeaders().get(KafkaHeaders.OFFSET, Long.cl
                return null;
            }

        };
    }

}
```

(2)、批量消费异常处理器

```
@Service
public class ConsumerService {

    /**
     * 消息监听器
     */
    @KafkaListener( topics = {"test"},groupId = "group1",errorHandler = "listenErrorH
    public void listen(List<String> messages) {
        for(String msg:messages){
            System.out.println(msg);
        }
        // 创建异常，触发异常处理器
        throw new NullPointerException("测试错误处理器");
    }

    /**
     * 异常处理器
     */
    @Bean
    public ConsumerAwareListenerErrorHandler listenErrorHandler() {
        return new ConsumerAwareListenerErrorHandler() {

            @Override
            public Object handleError(Message<?> message,
                                      ListenerExecutionFailedException e,
                                      Consumer<?, ?> consumer) {
                System.out.println("message:" + message.getPayload());
                System.out.println("exception:" + e.getMessage());
                consumer.seek(new TopicPartition(message.getHeaders().get(KafkaHeader
                              message.getHeaders().get(KafkaHeaders.RECEIVED_PARTIT
                              message.getHeaders().get(KafkaHeaders.OFFSET, Long.cl
                return null;
            }

        };
    }

}
```

(3)、全局异常处理

将异常处理器添加到 kafkaListenerContainerFactory 中来设置全局异常处理。

```
@Configuration
@EnableKafka
public class ConsumerConfigDemo {
    @Bean
    public Map<String, Object> consumerConfigs() {
        Map<String, Object> propsMap = new HashMap<>();
        propsMap.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "127.0.0.1:9092");
        propsMap.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, true);
        propsMap.put(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG, "100");
        propsMap.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer
        propsMap.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializ
        return propsMap;
```

```
    @Bean
    public ConsumerFactory<Integer, String> consumerFactory() {
        return new DefaultKafkaConsumerFactory<>(consumerConfigs());
    }

    @Bean
    KafkaListenerContainerFactory<ConcurrentMessageListenerContainer<Integer, String>
        ConcurrentKafkaListenerContainerFactory<Integer, String>
                factory = new ConcurrentKafkaListenerContainerFactory<>();
        factory.setConsumerFactory(consumerFactory());
        factory.setConcurrency(3);
        factory.getContainerProperties().setPollTimeout(3000);
        // 将单条消息异常处理器添加到参数中
        factory.setErrorHandler(errorHandler);
        // 将批量消息异常处理器添加到参数中
        //factory.setErrorHandler(errorHandler);
        return factory;
    }

    /**
     * 单消息消费异常处理器
     */
    @Bean
    public ConsumerAwareListenerErrorHandler listenErrorHandler() {
        return new ConsumerAwareListenerErrorHandler() {

            @Override
            public Object handleError(Message<?> message,
                                      ListenerExecutionFailedException e,
                                      Consumer<?, ?> consumer) {
                System.out.println("message:" + message.getPayload());
                System.out.println("exception:" + e.getMessage());
                consumer.seek(new TopicPartition(message.getHeaders().get(KafkaHeader
                              message.getHeaders().get(KafkaHeaders.RECEIVED_PARTIT
                        message.getHeaders().get(KafkaHeaders.OFFSET, Long.class));
                return null;
            }

        };
    }

    /**
     * 批量息消费异常处理器
     */
    @Bean
    public ConsumerAwareListenerErrorHandler listenErrorHandler() {
        return new ConsumerAwareListenerErrorHandler() {

            @Override
            public Object handleError(Message<?> message,
                                      ListenerExecutionFailedException e,
                                      Consumer<?, ?> consumer) {
                System.out.println("message:" + message.getPayload());
                System.out.println("exception:" + e.getMessage());
                consumer.seek(new TopicPartition(message.getHeaders().get(KafkaHeader
                              message.getHeaders().get(KafkaHeaders.RECEIVED_PARTIT
                              message.getHeaders().get(KafkaHeaders.OFFSET, Long.cl
                return null;
            }

        };
    }

    /**
     *  监听消息，接收过滤器过滤后的消息
     */
    @KafkaListener(topics = {"test"},groupId = "group1")
    public void kafkaListener(String message){
        System.out.println("消息: "+message);
    }

}
```

10、Kafka Consumer 手动/自动提交 Offset

在kafka的消费者中有一个非常关键的机制，那就是 offset 机制。它使得 Kafka 在消费的过程中即使挂了或者引发再均衡问题重新分配 Partation，当下次重新恢复消费时仍然可以知道从哪里开始消费。

Kafka中偏移量的自动提交是由参数 `enable_auto_commit` 和 `auto_commit_interval_ms` 控制的，当 `enable_auto_commit=true` 时，Kafka在消费的过程中会以频率为 `auto_commit_interval_ms` 向 Kafka 自带的 topic(__consumer_offsets) 进行偏移量提交，具体提交到哪个 Partation 是以算法：”partation=hash(group_id)%50” 来计算的。

在 Spring 中对 Kafka 设置手动或者自动提交Offset如下：

(1)、自动提交

自动提交需要配置下面两个参数：

- auto.commit.enable=true：是否将offset维护交给kafka自动提交到zookeeper中维护，设置为true。
- auto.commit.interval.ms=10000：自动提交时间间隔。

配置示例如下：

```
@Configuration
@EnableKafka
public class ConsumerConfigDemo {
    @Bean
    public Map<String, Object> consumerConfigs() {
        Map<String, Object> propsMap = new HashMap<>();
        // ---设置自动提交Offset为true
        propsMap.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);
        propsMap.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "127.0.0.1:9092");
        propsMap.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer
        propsMap.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializ
        return propsMap;
    }

    @Bean
    public ConsumerFactory<Integer, String> consumerFactory() {
        return new DefaultKafkaConsumerFactory<>(consumerConfigs());
    }

    @Bean
    KafkaListenerContainerFactory<ConcurrentMessageListenerContainer<Integer, String>
        ConcurrentKafkaListenerContainerFactory<Integer, String> factory = new Concur
        factory.setConsumerFactory(consumerFactory());
        // 消费者线程数
        factory.setConcurrency(3);
        // 拉取超时时间
        factory.getContainerProperties().setPollTimeout(3000);
        return factory;
    }

    /**
     * -------------接收消息-------------
     */
    @KafkaListener(topics = {"test"}, groupId = "group1")
    public void kafkaListener(String message){
        System.out.println("消息: "+message);
    }
```

(2)、手动提交

- auto.commit.enable=false：是否将offset维护交给kafka自动提交到zookeeper中维护，设置为false。

然后需要在程序中设置ack模式,从而进行手动提交维护offset。

```
@Bean
KafkaListenerContainerFactory<ConcurrentMessageListenerContainer<Integer, String>> ka
    ConcurrentKafkaListenerContainerFactory<Integer, String> factory = new Concurrent
    factory.setConsumerFactory(consumerFactory());
    factory.setConcurrency(3);
    factory.getContainerProperties().setPollTimeout(3000);
    设置ACK模式(手动提交模式，这里有七种)
    factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.RECORD);
    return factory;
}
```

在 kafkaListenerContainerFactory 配置中设置 AckMode，它有七种模式分别为：

- **RECORD：** 每处理完一条记录后提交。
- **BATCH(默认)：** 每次poll一批数据后提交一次，频率取决于每次poll的调用频率。
- **TIME：** 每次间隔ackTime的时间提交。
- **COUNT：** 处理完poll的一批数据后并且距离上次提交处理的记录数超过了设置的ackCount就提交。
- **COUNT_TIME：** TIME和COUNT中任意一条满足即提交。
- **MANUAL：** 手动调用Acknowledgment.acknowledge()后，并且处理完poll的这批数据后提交。
- **MANUAL_IMMEDIATE：** 手动调用Acknowledgment.acknowledge()后立即提交。

> 注意：如果设置 AckMode 模式为 MANUAL 或者 MANUAL_IMMEDIATE，则需要对监听消息的方法中，引入 Acknowledgment 对象参数，并调用 acknowledge() 方法进行手动提交

手动提交下这里将列出七种ACK模式示例，如下：

- **ACK 模式：** RECORD
- **描述：** 每处理完一条记录后提交。

```
@Configuration
@EnableKafka
public class ConsumerConfigDemo {
    @Bean
    public Map<String, Object> consumerConfigs() {
        Map<String, Object> propsMap = new HashMap<>();
        // ---设置自动提交Offset为false
        propsMap.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);
        propsMap.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "127.0.0.1:9092");
        propsMap.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer
        propsMap.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializ
        return propsMap;
    }

    @Bean
    public ConsumerFactory<Integer, String> consumerFactory() {
        return new DefaultKafkaConsumerFactory<>(consumerConfigs());
    }

    @Bean
    KafkaListenerContainerFactory<ConcurrentMessageListenerContainer<Integer, String>
        ConcurrentKafkaListenerContainerFactory<Integer, String> factory = new Concur
        factory.setConsumerFactory(consumerFactory());
        factory.setConcurrency(3);
        factory.getContainerProperties().setPollTimeout(3000);
        // 设置ACK模式为RECORD
        factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.RECOR
        return factory;
    }

    /**
     * -------------接收消息-------------
     */
    @KafkaListener(topics = {"test"}, groupId = "group1")
    public void kafkaListener(String message){
        System.out.println("消息："+message);
    }
```

- **ACK 模式：** BATCH
- **描述：** 每次poll一批数据后提交一次，频率取决于每次poll的调用频率。

```
@Configuration
@EnableKafka
public class ConsumerConfigDemo {
    @Bean
    public Map<String, Object> consumerConfigs() {
        Map<String, Object> propsMap = new HashMap<>();
        // ---设置自动提交Offset为false
        propsMap.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);
        // 设置每次批量消费数目，例如生产者生成10条数据，设置此值为4，那么需要三次批消费（三次中每次
        propsMap.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, "4");
        propsMap.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "127.0.0.1:9092");
        propsMap.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer
        propsMap.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializ
        return propsMap;
    }

    @Bean
    public ConsumerFactory<Integer, String> consumerFactory() {
        return new DefaultKafkaConsumerFactory<>(consumerConfigs());
    }

    @Bean
    KafkaListenerContainerFactory<ConcurrentMessageListenerContainer<Integer, String>
        ConcurrentKafkaListenerContainerFactory<Integer, String> factory = new Concur
        factory.setConsumerFactory(consumerFactory());
        factory.setConcurrency(3);
        factory.getContainerProperties().setPollTimeout(3000);
        // 开启批量消费监听器
        factory.setBatchListener(true);
        // 设置ACK模式为BATCH
        factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.BATCH
        return factory;
    }

    /**
     * -------------接收消息-------------
     * 批量消费时，设置参数为List来接收数据
     */
    @KafkaListener(topics = {"test"}, groupId = "group1")
    public void kafkaListener(List<String> message){
        System.out.println("消息："+message);
    }
```

- **ACK 模式：** COUNT
- **描述：** 处理完poll的一批数据后并且距离上次提交处理的记录数超过了设置的ackCount值就提交。

```
@Configuration
@EnableKafka
public class ConsumerConfigDemo {
    @Bean
```

```
        // ---设置自动提交Offset为false
        propsMap.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);
        propsMap.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "127.0.0.1:9092");
        propsMap.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer
        propsMap.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializ
        return propsMap;
    }

    @Bean
    public ConsumerFactory<Integer, String> consumerFactory() {
        return new DefaultKafkaConsumerFactory<>(consumerConfigs());
    }

    @Bean
    KafkaListenerContainerFactory<ConcurrentMessageListenerContainer<Integer, String>
        ConcurrentKafkaListenerContainerFactory<Integer, String> factory = new Concur
        factory.setConsumerFactory(consumerFactory());
        factory.setConcurrency(3);
        factory.getContainerProperties().setPollTimeout(3000);
        // 设置ACK模式为COUNT
        factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.COUNT
        // 设置AckCount数目，每接收AckCount条记录数就提交Offset偏移量
        factory.getContainerProperties().setAckCount(10);
        return factory;
    }

    /**
     * -------------接收消息-------------
     */
    @KafkaListener(topics = {"test"}, groupId = "group1")
    public void kafkaListener(String message){
        System.out.println("消息："+message);
    }
```

- **ACK 模式：** TIME
- **描述：** 每次间隔ackTime的时间提交。

```
@Configuration
@EnableKafka
public class ConsumerConfigDemo {
    @Bean
    public Map<String, Object> consumerConfigs() {
        Map<String, Object> propsMap = new HashMap<>();
        // ---设置自动提交Offset为false
        propsMap.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);
        propsMap.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "127.0.0.1:9092");
        propsMap.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer
        propsMap.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializ
        return propsMap;
    }

    @Bean
    public ConsumerFactory<Integer, String> consumerFactory() {
        return new DefaultKafkaConsumerFactory<>(consumerConfigs());
    }

    @Bean
    KafkaListenerContainerFactory<ConcurrentMessageListenerContainer<Integer, String>
        ConcurrentKafkaListenerContainerFactory<Integer, String> factory = new Concur
        factory.setConsumerFactory(consumerFactory());
        factory.setConcurrency(3);
        factory.getContainerProperties().setPollTimeout(3000);
        // 设置ACK模式为TIME
        factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.TIME)
        // 设置提交Ack的时间间隔，单位(ms)
        factory.getContainerProperties().setAckTime(1000);
        return factory;
    }

    /**
     * -------------接收消息-------------
     */
    @KafkaListener(topics = {"test"}, groupId = "group1")
    public void kafkaListener(String message){
        System.out.println("消息："+message);
    }
```

- **ACK 模式：** COUNT_TIME。
- **描述：** 每次间隔ackTime的时间或处理完poll的一批数据后并且距离上次提交处理的记录数超过了设置的ackCount值，任意一条满足即提交。

```
@Configuration
@EnableKafka
public class ConsumerConfigDemo {
    @Bean
    public Map<String, Object> consumerConfigs() {
        Map<String, Object> propsMap = new HashMap<>();
        // ---设置自动提交Offset为false
        propsMap.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);
        propsMap.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "127.0.0.1:9092");
        propsMap.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer
        propsMap.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializ
        return propsMap;
    }

    @Bean
    public ConsumerFactory<Integer, String> consumerFactory() {
        return new DefaultKafkaConsumerFactory<>(consumerConfigs());
    }

    @Bean
    KafkaListenerContainerFactory<ConcurrentMessageListenerContainer<Integer, String>
        ConcurrentKafkaListenerContainerFactory<Integer, String> factory = new Concur
        factory.setConsumerFactory(consumerFactory());
        factory.setConcurrency(3);
        factory.getContainerProperties().setPollTimeout(3000);
        // 设置ACK模式为COUNT_TIME
        factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.COUNT
        // 设置提交Ack的时间间隔，单位(ms)
        factory.getContainerProperties().setAckTime(1000);
        // 设置AckCount数目，每接收AckCount条记录数就提交Offset偏移量
        factory.getContainerProperties().setAckCount(10);
        return factory;
    }

    /**
     * -------------接收消息-------------
     */
    @KafkaListener(topics = {"test"}, groupId = "group1")
    public void kafkaListener(String message){
        System.out.println("消息："+message);
    }
```

- **ACK 模式：** MANUAL
- **描述：** 手动调用Acknowledgment.acknowledge()后，并且处理完poll的这批数据后提交。

```
@Configuration
@EnableKafka
public class ConsumerConfigDemo {
    @Bean
```

```
        Map<String, Object> propsMap = new HashMap<>();
        // ---设置自动提交Of
        propsMap.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);
        // 设置每次批量消费数目，例如生产者生成10条数据，设置此值为4，那么需要三次批消费（三次中每次
        propsMap.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, "4");
        propsMap.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "127.0.0.1:9092");
        propsMap.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer
        propsMap.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializ
        return propsMap;
    }

    @Bean
    public ConsumerFactory<Integer, String> consumerFactory() {
        return new DefaultKafkaConsumerFactory<>(consumerConfigs());
    }

    @Bean
    KafkaListenerContainerFactory<ConcurrentMessageListenerContainer<Integer, String>
        ConcurrentKafkaListenerContainerFactory<Integer, String> factory = new Concur
        factory.setConsumerFactory(consumerFactory());
        factory.setConcurrency(3);
        factory.getContainerProperties().setPollTimeout(3000);
        // 开启批量消费监听器
        factory.setBatchListener(true);
        // 设置ACK模式为MANUAL
        factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.MANUA
        return factory;
    }

    /**
     * -------------接收消息-------------
     * 批量消费时，设置参数为List来接收数据，并且因为ack模式为MANUAL，所以需要手动调用acknowledg
     */
    @KafkaListener(topics = {"test"}, groupId = "group1")
    public void kafkaListener(List<String> message, Acknowledgment acknowledgment){
        System.out.println("消息："+message);
        // 手动执行acknowledge()提交offset偏移量
        acknowledgment.acknowledge();
    }
```

- **ACK 模式：** MANUAL_IMMEDIATE
- **描述：** 手动调用Acknowledgment.acknowledge()后立即提交。

```
@Configuration
@EnableKafka
public class ConsumerConfigDemo {
    @Bean
    public Map<String, Object> consumerConfigs() {
        Map<String, Object> propsMap = new HashMap<>();
        // ---设置自动提交Offset为false
        propsMap.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);
        // 设置每次批量消费数目，例如生产者生成10条数据，设置此值为4，那么需要三次批消费（三次中每次
        propsMap.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, "4");
        propsMap.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "127.0.0.1:9092");
        propsMap.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer
        propsMap.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializ
        return propsMap;
    }

    @Bean
    public ConsumerFactory<Integer, String> consumerFactory() {
        return new DefaultKafkaConsumerFactory<>(consumerConfigs());
    }

    @Bean
    KafkaListenerContainerFactory<ConcurrentMessageListenerContainer<Integer, String>
        ConcurrentKafkaListenerContainerFactory<Integer, String> factory = new Concur
        factory.setConsumerFactory(consumerFactory());
        factory.setConcurrency(3);
        factory.getContainerProperties().setPollTimeout(3000);
        // 开启批量消费监听器
        factory.setBatchListener(true);
        // 设置ACK模式为MANUAL_IMMEDIATE
        factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.MANUA
        return factory;
    }

    /**
     * -------------接收消息-------------
     * 批量消费时，设置参数为List来接收数据，并且因为ack模式为MANUAL，所以需要手动调用acknowledg
     */
    @KafkaListener(topics = {"test"}, groupId = "group1")
    public void kafkaListener(List<String> message, Acknowledgment acknowledgment){
        System.out.println("消息："+message);
        // 手动执行acknowledge()提交offset偏移量
        acknowledgment.acknowledge();
    }
}
```

Springboot2(30)集成kafka--详细讲解@KafkaListener_co... 4-4
操作Topic 配置 @ComponentpublicclassPrividerKafkaConfig{@Value("${spring.kafka.bootstrap-servers}")privateString bo…

Spring Boot集成Kafka 流水不腐小夏 1万+
Spring Boot集成Kafka Spring Boot集成Kafka 前提介绍 Kafka 简介 Topics & logs Distribution Producers Consumers Guara…

SpringBoot Kafka 整合使用 http://www.54tianzhisheng.cn/ 5万+
前提 假设你了解过 SpringBoot 和 Kafka。 1、SpringBoot 如果对 SpringBoot 不了解的话，建议去看看 DD 大佬 和 纯洁的…

SpringBoot Kafka工具类封装 地表最强菜鸡的博客 4745
bootstrap需要自己配置一下，其他的直接用就可以。 package com.oal.microservice.util; import com.alibaba.fastjson.JSON…

Apache Kafka-SpringBoot整合Kafka发送复杂对象 小工匠 617
文章目录Spring Kafka概述Code Spring Kafka概述 Spring提供了 Spring-Kafka 项目来操作 Kafka。 https://spring.io/project…

SpringBoot Kafka 整合使用 meng19910117的博客 959
kafka需要进行三次握手链接 转载自：https://blog.csdn.net/tzs_1041218129/article/details/78988439（亲测） https://blog.c…

springboot 集成kafka系列 一、安装kafka yfz792178428的专栏 868
环境介绍 操作系统win10、jdk1.8、 1.安装并启动zookeeper 1.1官网https://zookeeper.apache.org/releases.html#download…

Spring Boot 2.x 最佳实践之Spring for Apache Kafka集成 人在码途，逐日拾光 1386
Spring Boot 2.x 最佳实践之Spring for Apache Kafka集成 原文：https://blog.csdn.net/hadues/article/details/88974967 这篇…

SpringBoot集成Kafka实现消息的生产和消费 weixin_44056249的博客 575
SpringBoot集成Kafka实现消息的生产和消费KafkaSpringBoot实现Kafka生产者 Kafka Kafka作为一款优异的消息中间件以…

SpringBoot整合Kafka 今天又是充满希望的一天 1223
一：环境准备. 1.1 Linux云服务器上安装Zookeeper,Kafka.可以参照我的这两篇博客.Zookeeper,Kafka.跟着做,一遍过. （…

使用spring-kafka操作kafka 山鹰的专栏 1万+
添加依赖 org.springframework.kafka spring-kafka 1.1.1.RELEASE 消息生产者 消息生产者spring配置 spring-producer.xml …