Spring Kafka 2.3.4 参考手册

大眼鱼叔叔

目　录

Spring Kafka 2.3.4 参考手册

1. 前言

原文：https://docs.spring.io/spring-kafka/docs/2.3.4.RELEASE/reference/html/#preface

Spring Kafka 是针对 Kafka 的开发框架。它提供了一个模板用于发送消息，也可以支持基于消息驱动的 POJO。

2. 变更内容

原文：https://docs.spring.io/spring-kafka/docs/2.3.4.RELEASE/reference/html/#whats-new-part

本节涵盖了从 2.2 版到 2.3 版的变更内容。另请参阅 Spring Integration for Apache Kafka（3.2 版）的变更内容。

2.1.1. 提示，技巧和示例

添加了一个新的章节的提示，技巧和示例。

2.1.2. Kafka Client 版本

kafka-clients 要求 2.3.0 及以上版本。

2.1.3. Class/Package 变更

TopicPartitionInitialOffset 已经过期，请使用 TopicPartitionOffset 替代。

2.1.4. Configuration 变更

从版本 2.3.4 开始，missingTopicsFatal 容器属性默认改为 flase。如果为 true，则当代理（Broker）宕机时应用程序将启动失败，很多用户将受此变更的影响。鉴于 Kafka 是一个高可用的平台，我们认为在代理（Broker）全部宕机的情况下启动应用程序并不是常见的情况。

2.1.5. Producer 和 Consumer Factory 变更

The DefaultKafkaProducerFactory can now be configured to create a producer per thread. You can also provide Supplier instances in the constructor as an alternative to either configured classes (which require no-arg constructors), or constructing with Serializer instances, which are then shared between all Producers. See Using DefaultKafkaProducerFactory for more information.
The same option is available with Supplier instances in DefaultKafkaConsumerFactory. See Using KafkaMessageListenerContainer for more information.

2.1.6. Listener Container 变更

Previously, error handlers received ListenerExecutionFailedException (with the actual listener exception as the cause) when the listener was invoked using a listener adapter (such as @KafkaListener s). Exceptions thrown by native GenericMessageListener s were passed to the error handler unchanged. Now a ListenerExecutionFailedException is always the argument (with the actual listener exception as the cause), which provides access to the container’s group.id property.

Because the listener container has it’s own mechanism for committing offsets, it prefers the Kafka ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG to be false. It now sets it to false automatically unless specifically set in the consumer factory or the container’s consumer property overrides.

The ackOnError property is now false by default. See Seek To Current Container Error Handlers for more information.

It is now possible to obtain the consumer’s group.id property in the listener method. See Obtaining the Consumer group.id for more information.

The container has a new property recordInterceptor allowing records to be inspected or modified before invoking the listener. A CompositeRecordInterceptor is also provided in case you need to invoke multiple interceptors. See Message Listener Containers for more information.

The ConsumerSeekAware has new methods allowing you to perform seeks relative to the beginning, end, or current position and to seek to the first offset greater than or equal to a time stamp. See Seeking to a Specific Offset for more information.

A convenience class AbstractConsumerSeekAware is now provided to simplify seeking. See Seeking to a Specific Offset for more information.

The ContainerProperties provides an idleBetweenPolls option to let the main loop in the listener container to sleep between KafkaConsumer.poll() calls. See its JavaDocs and Using KafkaMessageListenerContainer for more information.

When using AckMode.MANUAL (or MANUAL_IMMEDIATE) you can now cause a redelivery by calling nack on the Acknowledgment. See Committing Offsets for more information.

Listener performance can now be monitored using Micrometer Timer s. See Monitoring Listener Performance for more information.

The containers now publish additional consumer lifecyle events relating to startup. See Application Events for more information.

Transactional batch listeners can now support zombie fencing. See Transactions for more information.

The listener container factory can now be configured with a ContainerCustomizer to further configure each container after it has been created and configured. See Container factory for more information.

2.1.7. ErrorHandler 变更

The SeekToCurrentErrorHandler now treats certain exceptions as fatal and disables retry for those, invoking the recoverer on first failure.
The SeekToCurrentErrorHandler and SeekToCurrentBatchErrorHandler can now be configured to apply a BackOff (thread sleep) between delivery attempts.
Starting with version 2.3.2, recovered records' offsets will be committed when the error handler returns after recovering a failed record.
See Seek To Current Container Error Handlers for more information.
The DeadLetterPublishingRecoverer, when used in conjunction with an ErrorHandlingDeserializer2, now sets the payload of the message sent to the dead-letter topic, to the original value that could not be deserialized. Previously, it was null and user code needed to extract the DeserializationException from the message headers. See Publishing Dead-letter Records for more information.

2.1.8. TopicBuilder

提供了一个新类 `TopicBuilder` ，可以更方便地创建 `NewTopic` `@Bean` 。 有关更多信息，请参见配置主题。

2.1.9. Kafka Streams 变更

现在，您可以对 `@EnableKafkaStreams` 创建的 `StreamsBuilderFactoryBean` 进行额外的配置。 有关更多信息，请参见 流配置。
新提供了 `RecoveringDeserializationExceptionHandler` ，它允许恢复具有反序列化错误的记录。 可以将其与 `DeadLetterPublishingRecoverer` 结合使用，以将这些记录发送到死信主题。 有关更多信息，请参见 从反序列化异常中恢复。
提供了 `HeaderEnricher` 转换器（Transformer），使用 SpEL 生成标头值。 有关更多信息，请参见 Header Enricher。
已提供 `MessagingTransformer` 。 这允许 Kafka 流拓扑与 Spring 消息组件（例如 Spring Integration flow）进行交互。 有关更多信息，请参见 MessagingTransformer 和从 KStream 调用 Spring Integration flow。

2.1.10. JSON Component 变更

Now all the JSON-aware components are configured by default with a Jackson ObjectMapper

produced by the JacksonUtils.enhancedObjectMapper(). The JsonDeserializer now provides TypeReference-based constructors for better handling of target generic container types. Also a JacksonMimeTypeModule has been introduced for serialization of org.springframework.util.MimeType to plain string. See its JavaDocs and Serialization, Deserialization, and Message Conversion for more information.
A ByteArrayJsonMessageConverter has been provided as well as a new super class for all Json converters, JsonMessageConverter. Also, a StringOrBytesSerializer is now available; it can serialize byte[], Bytes and String values in ProducerRecord s. See Spring Messaging Message Conversion for more information.
The JsonSerializer, JsonDeserializer and JsonSerde now have fluent APIs to make programmatic configuration simpler. See the javadocs, Serialization, Deserialization, and Message Conversion, and Streams JSON Serialization and Deserialization for more informaion.

2.1.11. ReplyingKafkaTemplate

When a reply times out, the future is completed exceptionally with a KafkaReplyTimeoutException instead of a KafkaException.
此外，现在提供了重载的 `sendAndReceive` 方法，该方法允许在每个消息的基础上指定回复超时事件。

2.1.12. AggregatingReplyingKafkaTemplate

通过聚合来自多个接收者的回复，扩展了 `ReplyingKafkaTemplate` 。 有关更多信息，请参见 聚合多个回复。
Extends the ReplyingKafkaTemplate by aggregating replies from multiple receivers. See Aggregating Multiple Replies for more information.

2.1.13. 事务变更

现在，您可以在 `KafkaTemplate` 和 `KafkaTransactionManager` 上覆盖生产者工厂的 `transactionIdPrefix` 。 有关更多信息，请参见 transactionIdPrefix。
You can now override the producer factory’ s transactionIdPrefix on the KafkaTemplate and KafkaTransactionManager. See transactionIdPrefix for more information.

2.1.14. New Delegating Serializer/Deserializer

提供了一个委派的序列化器和反序列化器，利用标头来启用生产和消费记录（使用多种键/值类型）。 有关更多信息，请参见 委派序列化器和反序列化器。

2.1.15. New Retrying Deserializer

新增了一个 `RetryingDeserializer` ，用于发生瞬时错误时（比如可能发生的网络问题）重试序列化
有关更多信息，请参见 Retrying Deserializer。

2.1.16. New function for recovering from deserializing errors

ErrorHandlingDeserializer2 now uses a POJO (FailedDeserializationInfo) for passing all the contextual information around a deserialization error. This enables the code to access to extra information that was missing in the old BiFunction<byte[], Headers, T> failedDeserializationFunction.

2.1.17. EmbeddedKafkaBroker 变更

现在，您可以在注解中覆盖默认代理（Broker）列表属性名称。 有关更多信息，请参见 @EmbeddedKafka 注解 或 EmbeddedKafkaBroker Bean。

2.1.18. ReplyingKafkaTemplate 变更

现在，您可以自定义标题名称以进行关联，回复主题和回复分区。 有关更多信息，请参见 使用 ReplyingKafkaTemplate。

2.1.19. Header Mapper 变更

`DefaultKafkaHeaderMapper` 不再将简单的字符串值标头编码为JSON。

3. 介绍

原文：https://docs.spring.io/spring-kafka/docs/2.3.4.RELEASE/reference/html/#introduction

本章是对 Spring Kafka 的概述，包括一些基本概念和代码片段，以帮助读者快速入门。

3.1 快速开始

原文：https://docs.spring.io/spring-kafka/docs/2.3.4.RELEASE/reference/html/#quick-tour

这是一个五分钟的快速入门示例，带你开启 Spring Kafka 之旅。

先决条件：你必须先安装并运行 Kafka。然后，你必须获取 spring-kafka 的 jar 包及其所有依赖项。最简单的方式是在构建工具中声明一个依赖项。

Maven 依赖：

```
<dependency>
  <groupId>org.springframework.kafka</groupId>
  <artifactId>spring-kafka</artifactId>
  <version>2.3.4.RELEASE</version>
</dependency>
```

Gradle 依赖：

```
compile 'org.springframework.kafka:spring-kafka:2.3.4.RELEASE'
```

> 注意：当使用 Spring Boot 时，请忽略版本号，Boot 将自动引入正确版本号。

Maven 依赖：

```
<dependency>
  <groupId>org.springframework.kafka</groupId>
  <artifactId>spring-kafka</artifactId>
</dependency>
```

Gradle 依赖：

```
compile 'org.springframework.kafka:spring-kafka'
```

3.1.1 兼容性

3.1.1 兼容性

此文档适用于以下版本：

- Apache Kafka Clients 2.2.0
- Spring Framework 5.2.x
- 最低 Java 版本：8

3.1.2 一个非常简单的示例

3.1.2 一个非常简单的示例

如下所示，您可以使用纯 Java 来发送和接收消息：

```
@Test
public void testAutoCommit() throws Exception {
    logger.info("Start auto");
    ContainerProperties containerProps = new ContainerProperties("topic1", "top
ic2");
    final CountDownLatch latch = new CountDownLatch(4);
    containerProps.setMessageListener(new MessageListener<Integer, String>() {
        @Override
        public void onMessage(ConsumerRecord<Integer, String> message) {
            logger.info("received: " + message);
            latch.countDown();
        }
    });

    KafkaMessageListenerContainer<Integer, String> container = createContainer(
containerProps);
    container.setBeanName("testAuto");
    container.start();
    Thread.sleep(1000); // wait a bit for the container to start

    KafkaTemplate<Integer, String> template = createTemplate();
    template.setDefaultTopic(topic1);
    template.sendDefault(0, "foo");
    template.sendDefault(2, "bar");
    template.sendDefault(0, "baz");
    template.sendDefault(2, "qux");
    template.flush();

    assertTrue(latch.await(60, TimeUnit.SECONDS));
    container.stop();
    logger.info("Stop auto");
}
```

```
private KafkaMessageListenerContainer<Integer, String> createContainer(
                        ContainerProperties containerProps) {
    Map<String, Object> props = consumerProps();
    DefaultKafkaConsumerFactory<Integer, String> cf =
                            new DefaultKafkaConsumerFactory<Integer, String>(pr
ops);
    KafkaMessageListenerContainer<Integer, String> container =
```

```
                                new KafkaMessageListenerContainer<>(cf, containerPr
ops);
    return container;
}

private KafkaTemplate<Integer, String> createTemplate() {
    Map<String, Object> senderProps = senderProps();
    ProducerFactory<Integer, String> pf =
              new DefaultKafkaProducerFactory<Integer, String>(senderProps);
    KafkaTemplate<Integer, String> template = new KafkaTemplate<>(pf);
    return template;
}
```

```
private Map<String, Object> consumerProps() {
    Map<String, Object> props = new HashMap<>();
    props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
    props.put(ConsumerConfig.GROUP_ID_CONFIG, group);
    props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, true);
    props.put(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG, "100");
    props.put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, "15000");
    props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, IntegerDeserializer
.class);
    props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserialize
r.class);
    return props;
}
```

以上代码用于配置消费者客户端参数。

- ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG：指定连接 Kafka 集群所需的 broker 地址清单。
- GROUP_ID_CONFIG：此消费者所隶属的消费组的唯一标识，即消费组的名称。
- ENABLE_AUTO_COMMIT_CONFIG：配置是否开启自动提交消费位移的功能，默认开启。
- AUTO_COMMIT_INTERVAL_MS_CONFIG：当 enable.auto.commit 参数设置为 true 时才生效，表示开启自动提交消费位移功能时自动提交消费位移的时间间隔
- SESSION_TIMEOUT_MS_CONFIG：消费组管理协议中用来检测消费者是否失效的超时时间。
- KEY_DESERIALIZER_CLASS_CONFIG：消息中 key 所对应的反序列化类，需要实现 org.apache.kafka.common.serialization.Deserializer 接口。
- VALUE_DESERIALIZER_CLASS_CONFIG：消息中 value 所对应的反序列化类，需要实现 org.apache.kafka.common.serialization.Deserializer 接口。

```
private Map<String, Object> senderProps() {
    Map<String, Object> props = new HashMap<>();
    props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
    props.put(ProducerConfig.RETRIES_CONFIG, 0);
```

```
    props.put(ProducerConfig.BATCH_SIZE_CONFIG, 16384);
    props.put(ProducerConfig.LINGER_MS_CONFIG, 1);
    props.put(ProducerConfig.BUFFER_MEMORY_CONFIG, 33554432);
    props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, IntegerSerializer.cla
ss);
    props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.cl
ass);
    return props;
}
```

3.1.3 使用 Java 配置

3.1.3 使用 Java 配置

您可以使用 Java 中的 Spring Configuration 完成与上一个示例中相同的工作。 以下示例显示了如何执行此操作：

```
@Autowired
private Listener listener;
@Autowired
private KafkaTemplate<Integer, String> template;

@Test
public void testSimple() throws Exception {
    template.send("annotated1", 0, "foo");
    template.flush();
    assertTrue(this.listener.latch1.await(10, TimeUnit.SECONDS));
}

@Configuration
@EnableKafka
public class Config {

    @Bean
    ConcurrentKafkaListenerContainerFactory<Integer, String>
                        kafkaListenerContainerFactory() {
        ConcurrentKafkaListenerContainerFactory<Integer, String> factory =
                                new ConcurrentKafkaListenerContainerFactory<>()
;
        factory.setConsumerFactory(consumerFactory());
        return factory;
    }

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

    @Bean
    public Map<String, Object> consumerConfigs() {
        Map<String, Object> props = new HashMap<>();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, embeddedKafka.getBro
kersAsString());
        ...
        return props;
    }
```

```
    @Bean
    public Listener listener() {
        return new Listener();
    }

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

    @Bean
    public Map<String, Object> producerConfigs() {
        Map<String, Object> props = new HashMap<>();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, embeddedKafka.getBro
kersAsString());
        ...
        return props;
    }

    @Bean
    public KafkaTemplate<Integer, String> kafkaTemplate() {
        return new KafkaTemplate<Integer, String>(producerFactory());
    }

}
```

```
public class Listener {

    private final CountDownLatch latch1 = new CountDownLatch(1);

    @KafkaListener(id = "foo", topics = "annotated1")
    public void listen1(String foo) {
        this.latch1.countDown();
    }

}
```

3.1.4 使用 Spring Boot

3.1.4 使用 Spring Boot

Spring Boot 可以使事情变得更加简单。 以下 Spring Boot 应用程序将三个消息发送到一个主题，接收它们，然后停止。

```
@SpringBootApplication
public class Application implements CommandLineRunner {
    public static Logger logger = LoggerFactory.getLogger(Application.class);

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args).close();
    }

    @Autowired
    private KafkaTemplate<String, String> template;
    private final CountDownLatch latch = new CountDownLatch(3);

    @Override
    public void run(String... args) throws Exception {
        this.template.send("myTopic", "foo1");
        this.template.send("myTopic", "foo2");
        this.template.send("myTopic", "foo3");
        latch.await(60, TimeUnit.SECONDS);
        logger.info("All received");
    }

    @KafkaListener(topics = "myTopic")
    public void listen(ConsumerRecord<?, ?> cr) throws Exception {
        logger.info(cr.toString());
        latch.countDown();
    }
}
```

Boot 负责大多数配置。当我们使用本地代理时，我们需要的唯一的配置如下：

例子1. application.properties

```
spring.kafka.consumer.group-id=foo
spring.kafka.consumer.auto-offset-reset=earliest
```

配置解析：

- group-id：此消费者所隶属的消费组的唯一标识，即消费组的名称。
- auto-offset-reset：在 Kafka 中每当消费者查找不到所记录的消费位移时，就会根据消费者客户端参数 auto.offset.reset 的配置来决定如何从开始进行消费，这个参数的默认值为 “latest” ，表示从分区末尾开始消费消息。如果将 auto.offset.reset 参数配置为“earliest” ，那么消费者会从起始处，也就是 0 开始消费。

4. 参考

原文：https://docs.spring.io/spring-kafka/docs/2.3.4.RELEASE/reference/html/#reference

本章详细介绍了组成 Spring Kafka 的各种组件，以及使用 Spring 开发 Kafka 应用程序的核心类。

4.1 使用 Spring Kafka

原文：https://docs.spring.io/spring-kafka/docs/2.3.4.RELEASE/reference/html/#kafka

本节详细说明了使用 Spring Kafka 的各种问题。

4.1.1 配置主题

4.1.1 配置主题（Configuring Topics）

如果您在应用程序上下文中定义了一个 `KafkaAdmin` Bean，则它可以自动将主题（Topic）添加到代理（Broker）。 为此，您可以将每个主题的 `NewTopic` `@Bean` 添加到应用程序上下文中。 版本 2.3 引入了新类 `TopicBuilder` ，以使 Bean 的创建更加方便。 以下示例显示了如何执行此操作：

```
@Bean
public KafkaAdmin admin() {
    Map<String, Object> configs = new HashMap<>();
    configs.put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG, ...);
    return new KafkaAdmin(configs);
}

@Bean
public NewTopic topic1() {
    return TopicBuilder.name("thing1")
            .partitions(10)
            .replicas(3)
            .compact()
            .build();
}

@Bean
public NewTopic topic2() {
    return TopicBuilder.name("thing2")
            .partitions(10)
            .replicas(3)
            .config(TopicConfig.COMPRESSION_TYPE_CONFIG, "zstd")
            .build();
}

@Bean
public NewTopic topic3() {
    return TopicBuilder.name("thing3")
            .assignReplicas(0, Arrays.asList(0, 1))
            .assignReplicas(1, Arrays.asList(1, 2))
            .assignReplicas(2, Arrays.asList(2, 0))
            .config(TopicConfig.COMPRESSION_TYPE_CONFIG, "zstd")
            .build();
}
```

> 注意：使用 Spring Boot 时，会自动注册一个 KafkaAdmin Bean，因此您只需要 NewTopic @Bean 即

> 可。

默认情况下，如果代理（Broker）不可用，则会记录一条消息，但是上下文会继续加载。 您可以以编程方式调用 Admin 的 initialize() 方法，以便稍后再试。 如果您希望这种情况被认为是致命的，请将 Admin 的 fatalIfBrokerNotAvailable 属性设置为 true。 这样上下文就无法初始化了。

> 如果代理（Broker）支持（1.0.0或更高版本），并且 Admin 发现现有主题的分区数少于
>
> NewTopic.numPartitions，则 Admin 会自动增加分区数。

要获得更多高级功能，可以直接使用 AdminClient。 以下示例显示了如何执行此操作： ~~~ @Autowired private KafkaAdmin admin; ... AdminClient client = AdminClient.create(admin.getConfig()); ... client.close(); ~~~

4.1.2 发送消息

4.1.3 接收消息

4.1.3 接收消息

可以通过配置 MessageListenerContainer 并提供消息侦听器，或使用 @KafkaListener 注解来接收消息。

消息监听器（Message Listeners）

使用消息侦听器容器时，必须提供一个侦听器以接收数据。 当前有八种支持的消息侦听器接口：

```
public interface MessageListener<K, V> {
    void onMessage(ConsumerRecord<K, V> data);
}

public interface AcknowledgingMessageListener<K, V> {
    void onMessage(ConsumerRecord<K, V> data, Acknowledgment acknowledgment);
}

public interface ConsumerAwareMessageListener<K, V> extends MessageListener<K, V
> {
    void onMessage(ConsumerRecord<K, V> data, Consumer<?, ?> consumer);
}

public interface AcknowledgingConsumerAwareMessageListener<K, V> extends Messag
eListener<K, V> {
    void onMessage(ConsumerRecord<K, V> data, Acknowledgment acknowledgment, Co
nsumer<?, ?> consumer);
}

public interface BatchMessageListener<K, V> {
    void onMessage(List<ConsumerRecord<K, V>> data);
}

public interface BatchAcknowledgingMessageListener<K, V> {
    void onMessage(List<ConsumerRecord<K, V>> data, Acknowledgment acknowledgme
nt);
}

public interface BatchConsumerAwareMessageListener<K, V> extends BatchMessageLi
stener<K, V> {
    void onMessage(List<ConsumerRecord<K, V>> data, Consumer<?, ?> consumer);
}

public interface BatchAcknowledgingConsumerAwareMessageListener<K, V> extends B
atchMessageListener<K, V> {
    void onMessage(List<ConsumerRecord<K, V>> data, Acknowledgment acknowledgme
```

```
nt, Consumer<?, ?> consumer);
}
```

1. 使用自动提交或容器管理的提交方式时，可使用 MessageListener 来处理 poll() 操作接收的单个 ConsumerRecord。
2. 使用手动提交方式时，可使用 AcknowledgingMessageListener 来处理 poll() 操作接收到的各个 ConsumerRecord。
3. 使用自动提交或容器管理的提交方式时，可使用 ConsumerAwareMessageListener 来处理 poll() 操作接收的单个 ConsumerRecord。提供了对 Consumer 对象的访问。
4. 使用手动提交方式时，可使用 AcknowledgingConsumerAwareMessageListener 来处理 poll() 操作接收到的各个 ConsumerRecord。提供了对Consumer对象的访问。
5. 使用自动提交或容器管理的提交方式时，可使用 BatchMessageListener 来处理 poll() 操作接收的所有 ConsumerRecord。使用此接口时，不支持 AckMode.RECORD，因为已为侦听器提供了完整的批处理。
6. 使用手动提交方式时，可使用 BatchAcknowledgingMessageListener 来处理 poll() 操作接收到的所有 ConsumerRecord。
7. 使用自动提交或容器管理的提交方式时，可使用 BatchConsumerAwareMessageListener 来处理 poll() 操作接收的所有 ConsumerRecord。使用此接口时，不支持AckMode.RECORD，因为已为侦听器提供了完整的批处理。提供了对Consumer对象的访问。
8. 使用手动提交方式时，可使用 BatchAcknowledgingConsumerAwareMessageListener 来处理 poll() 操作接收到的所有 ConsumerRecord。提供了对 Consumer 对象的访问。

> 注意：Consumer 对象是非线程安全的，你只能在调用监听器的线程上使用它。

消息侦听器容器（Message Listener Containers）

提供了两个 MessageListenerContainer 实现：

- KafkaMessageListenerContainer
- ConcurrentMessageListenerContainer
  KafkaMessageListenerContainer 在单个线程上接收来自所有主题/分区的所有消息。
  ConcurrentMessageListenerContainer 委托给1个或多个 KafkaMessageListenerContainer 来提供多线程使用。

从版本 2.2.7 开始，您可以将 RecordInterceptor 添加到侦听器容器中。 它将在调用侦听器之前被调用，以允许检查或修改记录。 如果拦截器返回 null，则不调用侦听器。 当侦听器为批处理侦听器时，拦截器不会被调用。
从 2.3 版开始，CompositeRecordInterceptor 可用于调用多个拦截器。

默认情况下，使用事务时，在事务启动后将调用拦截器。 从版本 2.3.4 开始，您可以将侦听器容器的 interceptBeforeTx 属性设置为在事务开始之前调用拦截器。
没有为批处理侦听器提供拦截器，因为 Kafka 已经提供了 ConsumerInterceptor。

使用 KafkaMessageListenerContainer

可用构造函数如下：

```
public KafkaMessageListenerContainer(ConsumerFactory<K, V> consumerFactory,
                    ContainerProperties containerProperties)

public KafkaMessageListenerContainer(ConsumerFactory<K, V> consumerFactory,
                    ContainerProperties containerProperties,
                    TopicPartitionInitialOffset... topicPartitions)
```

ConcurrentMessageListenerContainer（稍后介绍）使用第二个构造函数在消费者实例之间分配 TopicPartitionOffset。ContainerProperties 包含主题和分区相关信息，构造函数如下：

```
public ContainerProperties(TopicPartitionInitialOffset... topicPartitions)

public ContainerProperties(String... topics)

public ContainerProperties(Pattern topicPattern)
```

- 第一个构造函数包含一个 TopicPartitionInitialOffset 数组参数，以明确指示容器使用哪个分区（使用 consumer 的 assign() 方法），并带有可选的初始偏移量：默认为正值； 默认情况下，负值相对于分区中的当前最后一个偏移量。
- 第二个构造函数包含一个字符串数据参数，Kafka 根据 group.id 属性分配分区（在整个组中分配分区）。
- 第三个构造函数使用正则表达式模式选择主题。

要将 MessageListener 分配给容器，请在创建 Container 时使用 ContainerProps.setMessageListener 方法：

```
ContainerProperties containerProps = new ContainerProperties("topic1", "topic2"
);
containerProps.setMessageListener(new MessageListener<Integer, String>() {
    ...
});
DefaultKafkaConsumerFactory<Integer, String> cf =
                        new DefaultKafkaConsumerFactory<>(consumerProps());
KafkaMessageListenerContainer<Integer, String> container =
                        new KafkaMessageListenerContainer<>(cf, containerProps)
;
```

```
return container;
```

注意，在创建 DefaultKafkaConsumerFactory 时，使用仅接受上述属性的构造函数意味着从配置中获取键和值的 Deserializer 类。 或者，可以将 Deserializer 实例传递给 DefaultKafkaConsumerFactory 构造函数以获取键或值，在这种情况下，所有消费者均共享相同的实例。 另一个选择是提供 Supplier （从版本2.3开始），该类将用于为每个消费者获取单独的 Deserializer 实例：

```
DefaultKafkaConsumerFactory<Integer, CustomValue> cf =
                        new DefaultKafkaConsumerFactory<>(consumerProps(), null
, () -> new CustomValueDeserializer());
KafkaMessageListenerContainer<Integer, String> container =
                        new KafkaMessageListenerContainer<>(cf, containerProps)
;
return container;
```

有关可设置的各种属性的更多信息，请参考 ContainerProperties 的 Javadoc 。
从 2.1.1 版开始，提供了一个名为 logContainerConfig 的新属性。 设为 true 并启用 INFO 日志记录后，每个侦听器容器（Listener Container）都会写入一条日志消息，以概述其配置属性。
默认情况下，主题偏移量提交的日志记录是使用 DEBUG 日志记录级别进行的。 从版本 2.1.2 开始，ContainerProperties 中有一个名为 commitLogLevel 的新属性，该属性可让您指定这些消息的日志级别。 例如，要将日志级别更改为 INFO，请使用 containerProperties.setCommitLogLevel(LogIfLevelEnabled.Level.INFO)。
从 2.2 版开始，添加了一个名为 missingTopicsFatal 的新容器属性（默认值：true）。 如果代理中没有任何已配置的主题，这将阻止容器启动。 如果侦听器配置为通过正则匹配，则该方法不适用。 以前，容器线程在 consumer.poll() 方法内循环，等待主题出现，同时记录许多消息。 除了日志，没有迹象表明存在问题。 若要还原以前的行为，可以将属性设置为false。

使用ConcurrentMessageListenerContainer

构造函数如下：

```
public ConcurrentMessageListenerContainer(ConsumerFactory<K, V> consumerFactory
,
                            ContainerProperties containerProperties)
```

它具有 concurrency 属性，例如 container.setConcurrency(3) 将创建3个 KafkaMessageListenerContainer。 Kafka 将使用其消费组管理功能在消费者之间分配分区。

> 注意：在监听多个主题时，默认的分区分布可能不是您期望的。 例如，如果您有3个主题，每个主题有5个分

> 区，而您想使用 concurrency = 15，则只会看到5个活动使用者，每个消费者都为每个主题分配了一个分区，而其他10个消费者处于空闲状态。 这是因为默认的 Kafka PartitionAssignor 是 RangeAssignor（请参阅其 javadocs）。 对于这种情况，您可能需要考虑使用 RoundRobinAssignor，它将在所有使用者之间分配分区。 然后，将为每个消费者分配一个主题/分区。 要更改 PartitionAssignor，请在提供给 DefaultKafkaConsumerFactory 的属性中设置partition.assignment.strategy 使用者属性（ConsumerConfigs.PARTITION_ASSIGNMENT_STRATEGY_CONFIG）。
>
> 使用 Spring Boot 时：
>
> `spring.kafka.consumer.properties.partition.assignment.strategy = org.apache.kaf`

对于第二个构造函数，ConcurrentMessageListenerContainer 在委托 KafkaMessageListenerContainer 中分配TopicPartition。

例如，如果提供了6个 TopicPartition，并且并发为3，则为0。 每个容器将获得2个分区。 对于5个 TopicPartition，两个容器将获得2个分区，第三个容器将得到1。如果并发大于 TopicPartitions 的数量，则并发性将向下调整，以便每个容器将获得一个分区。

注意：client.id属性（如果已设置）将附加-n，其中n是根据并发性使用的消费者实例。 启用JMX时，必须为MBean提供唯一的名称。

从1.3版开始，MessageListenerContainer提供了对基础KafkaConsumer指标的访问。 对于ConcurrentMessageListenerContainer而言，metrics（）方法将返回所有目标KafkaMessageListenerContainer实例的度量。 度量标准分为Map 。

@KafkaListener 注解

@KafkaListener 注解为简单的 POJO 侦听器提供了一种机制：

```
public class Listener {
    @KafkaListener(id = "foo", topics = "myTopic", clientIdPrefix = "myClientId
")
    public void listen(String data) {
        ...
    }
}
```

此机制需要在 @Configuration 类之一上使用 @EnableKafka 批注，以及用于配置基础 ConcurrentMessageListenerContainer 的侦听器容器工厂：默认情况下，应使用名称为 kafkaListenerContainerFactory 的 bean。

```
@Configuration
@EnableKafka
public class KafkaConfig {
```

```
    @Bean
    KafkaListenerContainerFactory<ConcurrentMessageListenerContainer<Integer, S
tring>>
                        kafkaListenerContainerFactory() {
        ConcurrentKafkaListenerContainerFactory<Integer, String> factory =
                                new ConcurrentKafkaListenerContainerFactory<>()
;
        factory.setConsumerFactory(consumerFactory());
        factory.setConcurrency(3);
        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> props = new HashMap<>();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, embeddedKafka.getBro
kersAsString());
        ...
        return props;
    }
}
```

请注意，要设置 container 属性，必须在工厂上使用 getContainerProperties() 方法。

从 2.1.1 版本开始，现在可以为创建注解的消费者设置 client.id 属性。 clientIdPrefix 带有 -n 后缀，其中 n 是表示使用并发时的容器号的整数。

您还可以为 POJO 侦听器配置明确的主题和分区（以及可选的初始偏移量）：

```
@KafkaListener(id = "bar", topicPartitions =
        { @TopicPartition(topic = "topic1", partitions = { "0", "1" }),
          @TopicPartition(topic = "topic2", partitions = "0",
             partitionOffsets = @PartitionOffset(partition = "1", initialOffset
 = "100"))
        })
public void listen(ConsumerRecord<?, ?> record) {
    ...
}
```

可以在 partitions 或 partitionOffsets 属性中指定分区，但不能在两个属性中同时指定分区。

当使用手动 AckMode 时，还可以向侦听器提供该 Acknowledgment。 此示例还显示了如何使用其他容器工厂。

```
@KafkaListener(id = "baz", topics = "myTopic",
          containerFactory = "kafkaManualAckListenerContainerFactory")
public void listen(String data, Acknowledgment ack) {
    ...
    ack.acknowledge();
}
```

最后，有关消息的元数据可从消息头获得，以下头名称可用于检索消息的头：

- KafkaHeaders.RECEIVED_MESSAGE_KEY
- KafkaHeaders.RECEIVED_TOPIC
- KafkaHeaders.RECEIVED_PARTITION_ID
- KafkaHeaders.RECEIVED_TIMESTAMP
- KafkaHeaders.TIMESTAMP_TYPE

```
@KafkaListener(id = "qux", topicPattern = "myTopic1")
public void listen(@Payload String foo,
        @Header(KafkaHeaders.RECEIVED_MESSAGE_KEY) Integer key,
        @Header(KafkaHeaders.RECEIVED_PARTITION_ID) int partition,
        @Header(KafkaHeaders.RECEIVED_TOPIC) String topic,
        @Header(KafkaHeaders.RECEIVED_TIMESTAMP) long ts
        ) {
    ...
}
```

从版本1.1开始，可以将 @KafkaListener 方法配置为处理整批消费者记录。 要将侦听器容器工厂配置为创建批处理侦听器，请设置 batchListener 属性：

```
@Bean
public KafkaListenerContainerFactory<?> batchFactory() {
    ConcurrentKafkaListenerContainerFactory<Integer, String> factory =
            new ConcurrentKafkaListenerContainerFactory<>();
    factory.setConsumerFactory(consumerFactory());
    factory.setBatchListener(true);
    return factory;
}
```

简单的批处理方式：

```
@KafkaListener(id = "list", topics = "myTopic", containerFactory = "batchFactor
y")
public void listen(List<String> list) {
    ...
```

```
}
```

通过配置主题，分区，偏移量等控制批处理的方式：

```
@KafkaListener(id = "list", topics = "myTopic", containerFactory = "batchFactor
y")
public void listen(List<String> list,
        @Header(KafkaHeaders.RECEIVED_MESSAGE_KEY) List<Integer> keys,
        @Header(KafkaHeaders.RECEIVED_PARTITION_ID) List<Integer> partitions,
        @Header(KafkaHeaders.RECEIVED_TOPIC) List<String> topics,
        @Header(KafkaHeaders.OFFSET) List<Long> offsets) {
    ...
}
```

通过 Message 对象控制批处理的方式：

```
@KafkaListener(id = "listMsg", topics = "myTopic", containerFactory = "batchFac
tory")
public void listen14(List<Message<?>> list) {
    ...
}

@KafkaListener(id = "listMsgAck", topics = "myTopic", containerFactory = "batch
Factory")
public void listen15(List<Message<?>> list, Acknowledgment ack) {
    ...
}
```

在这种情况下，不会对有效负载执行任何转换。

如果为 BatchMessagingMessageConverter 配置了 RecordMessageConverter，则还可以将通用类型添加到 Message 参数中，然后将转换有效负载。

您还可以收到 ConsumerRecord 对象的列表，但是它必须是在方法上定义的唯一参数（使用手动提交时，除了可选的Acknowledgment）。

```
@KafkaListener(id = "listCRs", topics = "myTopic", containerFactory = "batchFac
tory")
public void listen(List<ConsumerRecord<Integer, String>> list) {
    ...
}

@KafkaListener(id = "listCRsAck", topics = "myTopic", containerFactory = "batch
Factory")
public void listen(List<ConsumerRecord<Integer, String>> list, Acknowledgment a
ck) {
```

```
    ...
}
```

从 2.0 版开始，id 属性（如果存在）将用作 Kafka group.id 属性，并覆盖 Consumer Factory 中的已配置属性（如果存在）。 您还可以显式设置 groupId 或将 idIsGroup 设置为 false，以恢复使用使用者工厂 group.id 的先前行为。

您可以在注解属性中使用属性占位符或 SpEL 表达式，例如...

```
@KafkaListener(topics = "${some.property}")

@KafkaListener(topics = "#{someBean.someProperty}",
    groupId = "#{someBean.someProperty}.group")
```

从版本2.1.2开始，SpEL 表达式支持特殊的令牌 __listener，这是一个伪 bean 名称，表示此注解所在的当前 bean 实例。

例如，给定：

```
@Bean
public Listener listener1() {
    return new Listener("topic1");
}

@Bean
public Listener listener2() {
    return new Listener("topic2");
}
```

我们就可以使用：

```
public class Listener {
    private final String topic;

    public Listener(String topic) {
        this.topic = topic;
    }

    @KafkaListener(topics = "#{__listener.topic}",
        groupId = "#{__listener.topic}.group")
    public void listen(...) {
        ...
    }

    public String getTopic() {
        return this.topic;
```

```
    }
}
```

如果在不太可能的情况下有一个名为 __listener 的实际 bean，则可以使用 beanRef 属性更改表达式令牌...

```
@KafkaListener(beanRef = "__x", topics = "#{__x.topic}",
    groupId = "#{__x.topic}.group")
```

Rebalance Listeners

ContainerProperty 具有一个 ConsumerRebalanceListener 属性，该属性采用 Kafka 客户端的 ConsumerRebalanceListener 接口的实现。 如果未提供此属性，则容器将配置一个简单的日志侦听器，该日志侦听器在 INFO 级别下记录重新平衡事件。 该框架还添加了一个子接口 ConsumerAwareRebalanceListener：

```
public interface ConsumerAwareRebalanceListener extends ConsumerRebalanceListen
er {

    void onPartitionsRevokedBeforeCommit(Consumer<?, ?> consumer, Collection<To
picPartition> partitions);

    void onPartitionsRevokedAfterCommit(Consumer<?, ?> consumer, Collection<Top
icPartition> partitions);

    void onPartitionsAssigned(Consumer<?, ?> consumer, Collection<TopicPartition
> partitions);

}
```

请注意，撤销分区时有两个回调：第一个立即调用；第二个在提交任何未决的偏移量后调用。 如果您希望在某些外部存储库中保持偏移量，则这很有用。 例如：

```
containerProperties.setConsumerRebalanceListener(new ConsumerAwareRebalanceList
ener() {
    @Override
    public void onPartitionsRevokedBeforeCommit(Consumer<?, ?> consumer, Collec
tion<TopicPartition> partitions) {
        // acknowledge any pending Acknowledgments (if using manual acks)
    }

    @Override
    public void onPartitionsRevokedAfterCommit(Consumer<?, ?> consumer, Collect
ion<TopicPartition> partitions) {
```

```
            // ...
                store(consumer.position(partition));
            // ...
        }

        @Override
        public void onPartitionsAssigned(Collection<TopicPartition> partitions) {
            // ...
                consumer.seek(partition, offsetTracker.getOffset() + 1);
            // ...
        }
    });
```

过滤消息

在某些情况下，例如重平衡，可能会重新传递已处理的消息。框架无法知道是否已处理此类消息，即应用程序级功能。这被称为幂等接收器模式，Spring Integration 提供了其实现。

Spring Kafka 还通过 FilteringMessageListenerAdapter 类提供了一些帮助，该类可以包装您的 MessageListener。此类采用 RecordFilterStrategy 的实现，在该实现中，您将实现 filter 方法以发出消息重复消息并应将其丢弃的信号。它具有一个附加属性 ackDiscarded，该属性指示适配器是否应确认丢弃的记录；否则，它不可用。默认情况下为 false。

使用 @KafkaListener 时，请在容器工厂上设置 RecordFilterStrategy（以及可选的 ackDiscarded），并且侦听器将包装在适当的过滤适配器中。

此外，为使用批处理消息侦听器提供了 FilteringBatchMessageListenerAdapter。

有状态重试

重要的是要了解，上面讨论的重试会挂起使用者线程（如果使用 BackOffPolicy）；重试期间没有调用 Consumer.poll()。Kafka 有两个属性来判断消费者是否存活。 session.timeout.ms 用于确定使用者是否处于活动状态。由于 0.10.1.0 版本后的心跳测试是在后台线程上发送的，因此缓慢的消费者不再会对此产生影响。max.poll.interval.ms（默认为5分钟）用于确定使用者是否似乎被挂起（花费太长时间来处理上次轮询中的记录）。如果 poll() 之间的时间超过此配置，则代理将撤销分配的分区并执行重平衡。对于冗长的重试序列，这很容易发生。

从版本 2.1.3 开始，可以通过将有状态重试与 SeekToCurrentErrorHandler 结合使用来避免此问题。在这种情况下，每次传递尝试都将异常抛出回容器，并且错误处理程序将重新寻找未处理的偏移量，并且下一个poll() 将传递相同的消息。这样可以避免超出 max.poll.interval.ms 属性的问题（只要两次尝试之间的单个延迟不超过该时间）。因此，在使用 ExponentialBackOffPolicy 时，请务必确保 maxInterval 小于 max.poll.interval.ms 属性。要启用有状态重试，请使用 RetryingMessageListenerAdapter 构造函数，该构造函数接受有状态布尔参数

（将其设置为true）。使用侦听器容器工厂进行配置（对于@KafkaListener）时，请将工厂的 statefulRetry 属性设置为 true。

4.1.4 暂停/恢复侦听器容器

4.1.4 暂停/恢复侦听器容器

版本 2.1.3 向侦听器容器添加了 pause() 和 resume() 方法。以前，您可以在 ConsumerAwareMessageListener 中暂停使用者，然后通过侦听 ListenerContainerIdleEvent 来恢复它，以提供对 Consumer 对象的访问。尽管您可以通过事件侦听器将使用者放在空闲容器中，但在某些情况下，这不是线程安全的，因为无法保证在使用者线程上调用事件侦听器。为了安全地暂停/恢复使用者，您应该使用侦听器容器上的方法。 pause() 在下一个 poll() 之前生效；在当前poll() 返回之后，resume 才生效。容器暂停后，它将继续对使用者进行poll() ，如果正在使用组管理，则避免了重新平衡，但不会检索任何记录；有关更多信息，请参阅 Kafka 文档。

从版本 2.1.5 开始，可以调用 isPauseRequested() 来查看是否已调用 pause() 。但是，消费者可能尚未真正停下来。如果所有 Consumer 均已实际暂停，则 isConsumerPaused() 将返回 true。

此外，从 2.1.5 版本开始，ConsumerPausedEvent 和 ConsumerResumedEvent 都以容器作为源属性发布，而partitionPatition 则包含在 partitions 属性中。

5. 提示，技巧和例子

5.2 事务同步示例

5.2 事务同步示例

以下 Spring Boot 应用程序是同步数据库和 Kafka 事务的示例。

```
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

    @Bean
    public ApplicationRunner runner(KafkaTemplate<String, String> template) {
        return args -> template.executeInTransaction(t -> t.send("topic1", "tes
t"));
    }

    @Bean
    public ChainedKafkaTransactionManager<Object, Object> chainedTm(
            KafkaTransactionManager<String, String> ktm,
            DataSourceTransactionManager dstm) {

        return new ChainedKafkaTransactionManager<>(ktm, dstm);
    }

    @Bean
    public DataSourceTransactionManager dstm(DataSource dataSource) {
        return new DataSourceTransactionManager(dataSource);
    }

    @Bean
    public ConcurrentKafkaListenerContainerFactory<?, ?> kafkaListenerContainer
Factory(
            ConcurrentKafkaListenerContainerFactoryConfigurer configurer,
            ConsumerFactory<Object, Object> kafkaConsumerFactory,
            ChainedKafkaTransactionManager<Object, Object> chainedTM) {

        ConcurrentKafkaListenerContainerFactory<Object, Object> factory =
                new ConcurrentKafkaListenerContainerFactory<>();
        configurer.configure(factory, kafkaConsumerFactory);
        factory.getContainerProperties().setTransactionManager(chainedTM);
        return factory;
    }

    @Component
    public static class Listener {

        private final JdbcTemplate jdbcTemplate;
```

```
        private final KafkaTemplate<String, String> kafkaTemplate;

        public Listener(JdbcTemplate jdbcTemplate, KafkaTemplate<String, String>
 kafkaTemplate) {
            this.jdbcTemplate = jdbcTemplate;
            this.kafkaTemplate = kafkaTemplate;
        }

        @KafkaListener(id = "group1", topics = "topic1")
        public void listen1(String in) {
            this.kafkaTemplate.send("topic2", in.toUpperCase());
            this.jdbcTemplate.execute("insert into mytable (data) values ('" +
in + "')");
        }

        @KafkaListener(id = "group2", topics = "topic2")
        public void listen2(String in) {
            System.out.println(in);
        }
    }

    @Bean
    public NewTopic topic1() {
        return TopicBuilder.name("topic1").build();
    }

    @Bean
    public NewTopic topic2() {
        return TopicBuilder.name("topic2").build();
    }
}
```

```
spring.datasource.url=jdbc:mysql://localhost/integration?serverTimezone=UTC
spring.datasource.username=root
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.kafka.consumer.auto-offset-reset=earliest
spring.kafka.consumer.enable-auto-commit=false
spring.kafka.consumer.properties.isolation.level=read_committed

spring.kafka.producer.transaction-id-prefix=tx-

#logging.level.org.springframework.transaction=trace
#logging.level.org.springframework.kafka.transaction=debug
#logging.level.org.springframework.jdbc=debug
```

```
create table mytable (data varchar(20));
```