

# ProgramadorBuscado



## Episode 3 Spring for Apache Kafka acepta el mensaje

Etiquetas: [Spring for Apache Kafka](#) [Recibir mensaje](#) [Spring Kafka](#) [Aceptar el mensaje](#)

### Ni los famosos se libran de los kilos de más: así lucen ahora estos famosos

Malasaña

### Es desgarrador donde vive Ana Obregón a los 65 años

PsychicMonday

### Obtener un título de MBA en línea puede ser más fácil de lo que piensas

MBA online | Enlaces Publicitarios

Podemos aceptar el mensaje configurando uno `MessageListenerContainer` y proporcionar un escucha de mensajes o usando `@KafkaListener` anotaciones

## 3.1 Oyentes de mensajes

Cuando usamos un contenedor de escucha de mensajes, debemos proporcionar un escucha para aceptar los datos.

Actualmente hay ocho interfaces que admiten la escucha de mensajes. La siguiente es una lista de estas interfaces:

```
1 public interface MessageListener<K, V> {  
2  
3     void onMessage(ConsumerRecord<K, V> data);  
4  
5 }
```

Utilice esta interfaz para procesar cada uno recibido de una operación de encuesta de usuario de Kafka cuando use la confirmación automática o uno de los métodos de confirmación



**Clase A EQ Power**  
patrocinado por: Mercedes

LEE MAS

Intimidación

```
3 | void onMessage(ConsumerRecord<K, V> data, Acknowledgment acknowledgment);  
4 |  
5 | }
```

Use uno de estos métodos para manejar instancias individuales de `ConsumerRecord` recibidas de las operaciones de encuesta de usuarios de Kafka cuando use uno de los métodos de envío manual.

```
1 | public interface ConsumerAwareMessageListener<K, V> extends MessageListener<K, V>  
2 |  
3 |     void onMessage(ConsumerRecord<K, V> data, Consumer<?, ?> consumer);  
4 |  
5 | }
```

Use esta interfaz para procesar instancias individuales de `ConsumerRecord` recibidas de las operaciones de encuesta de usuarios de Kafka cuando se utiliza la confirmación automática o uno de los métodos de confirmación administrados por contenedor. Proporciona acceso al objeto `Consumer`.

```
1 | public interface AcknowledgingConsumerAwareMessageListener<K, V> extends MessageL  
2 |  
3 |     void onMessage(ConsumerRecord<K, V> data, Acknowledgment acknowledgment, Cons  
4 |  
5 | }
```

Use uno de estos métodos para manejar instancias individuales de `ConsumerRecord` recibidas de las operaciones de encuesta de usuarios de Kafka cuando use uno de los métodos de envío manual. Proporcionar acceso al objeto del consumidor

```
1 | public interface BatchMessageListener<K, V> {  
2 |  
3 |     void onMessage(List<ConsumerRecord<K, V>> data);  
4 |  
5 | }
```

Use esta interfaz para procesar todas las instancias de `ConsumerRecord` recibidas de las operaciones de sondeo de usuarios de Kafka cuando use la confirmación automática o uno de los métodos de confirmación administrados por contenedor. `AckMode.RECORD` no es compatible cuando se utiliza esta interfaz porque se proporciona un lote completo para el oyente.



```
1      public interface BatchAcknowledgingMessageListener<K, V> {  
2  
3      void onMessage(List<ConsumerRecord<K, V>> data, Acknowledgment acknowledgment  
4  
5      }
```

Use esta interfaz para procesar todas las instancias de `ConsumerRecord` recibidas de las operaciones de encuesta de usuarios de Kafka cuando use uno de los métodos de envío manual.

```
1      public interface BatchConsumerAwareMessageListener<K, V> extends BatchMessageList  
2  
3      void onMessage(List<ConsumerRecord<K, V>> data, Consumer<?, ?> consumer);  
4  
5      }
```

La acción cuando se utiliza la confirmación automática o uno de los métodos de envío administrados por contenedor. `AckMode.RECORD` no es compatible cuando se utiliza esta interfaz porque se proporciona un lote completo para el oyente. Proporciona acceso al objeto `Consumer`.

```
1      public interface BatchAcknowledgingConsumerAwareMessageListener<K, V> extends Bat  
2  
3      void onMessage(List<ConsumerRecord<K, V>> data, Acknowledgment acknowledgment  
4  
5      }
```

Use esta interfaz para procesar todas las instancias de `ConsumerRecord` recibidas de las operaciones de encuesta de usuarios de Kafka cuando use uno de los métodos de envío manual. Proporciona acceso al objeto `Consumer`.

Los objetos de consumo no son seguros para los hilos. Solo puede llamar a sus métodos en el hilo que llamó al oyente.

## 3.2 Contenedores de escucha de mensajes

Proporciona dos implementaciones de contenedor de escucha de mensajes

- `KafkaMessageListenerContainer`
- `ConcurrentMessageListenerContainer`

Intimidad

Este `KafkaMessageLisenterContainer` acepta todos los mensajes de todos los temas o particiones en un hilo.

Este proxy `ConcurrentMessageListenerContainer` acepta una o más instancias de `KafkaMessageListenerContainer` al proporcionar varios subprocesos.

### 3.2.1 Usando `KafkaMessageListenerContainer`

El constructor es el siguiente:

```
1 public KafkaMessageListenerContainer(ConsumerFactory<K, V> consumerFactory,  
2                                     ContainerProperties containerProperties)  
3  
4 public KafkaMessageListenerContainer(ConsumerFactory<K, V> consumerFactory,  
5                                     ContainerProperties containerProperties,  
6                                     TopicPartitionInitialOffset... topicPartitions)
```

Cada uno usa `ConsumerFactory` e información sobre el tema y la partición, así como otras configuraciones en el objeto `ContainerProperties`. `ConcurrentMessageListenerContainer` (descrito más adelante) usa el segundo constructor para distribuir `TopicPartitionInitialOffset` entre las instancias del consumidor. `ContainerProperties` tiene los siguientes constructores:

```
1 public ContainerProperties(TopicPartitionInitialOffset... topicPartitions)  
2  
3 public ContainerProperties(String... topics)  
4  
5 public ContainerProperties(Pattern topicPattern)
```

El primer constructor toma una matriz de parámetros `TopicPartitionInitialOffset` para indicar explícitamente qué particiones usa el contenedor (usando el método de asignación del consumidor) y un desplazamiento inicial opcional. Un valor positivo es el desplazamiento absoluto predeterminado. Por defecto, el valor negativo es relativo al último desplazamiento actual en la partición. Proporciona un constructor para `TopicPartitionInitialOffset` que acepta un argumento booleano adicional. Si esto es cierto, el desplazamiento inicial (positivo o negativo) es relativo a la ubicación actual del consumidor. El desplazamiento se aplica cuando se inicia el contenedor. El segundo usa una serie de temas, Kafka asigna particiones basadas en el atributo `group.id`: asigna particiones en todo el grupo. El tercero usa el patrón de expresión regular para seleccionar el tema.

Para asignar un `MessageListener` a un contenedor, puede usar el método `ContainerProps.setMessageListener` al crear un Contenedor. El siguiente ejemplo muestra cómo hacer esto:



```

1 | ContainerProperties containerProps = new ContainerProperties("topic1", "topic2");
2 |     containerProps.setMessageListener(new MessageListener<Integer, String>() {
3 |         ...
4 |     });
5 |     DefaultKafkaConsumerFactory<Integer, String> cf =
6 |         new DefaultKafkaConsumerFactory<Integer, String>(consumer
7 |             KafkaMessageListenerContainer<Integer, String> container =
8 |                 new KafkaMessageListenerContainer<>(cf, containerProps);
9 |         return container;

```

Para obtener más información sobre las diversas propiedades que puede establecer, consulte Javadoc para `ContainerProperties`.

A partir de la versión 2.1.1, puede usar una nueva propiedad llamada `logContainerConfig`. Si el registro verdadero e INFO está habilitado, cada contenedor de escucha escribe un mensaje de registro que registra sus propiedades de configuración.

De forma predeterminada, el registro de envíos de compensación de tema se realiza en el nivel de registro DEPURACIÓN. A partir de la versión 2.1.2, la propiedad denominada `commitLogLevel` en `ContainerProperties` le permite especificar el nivel de registro para estos mensajes. Por ejemplo, para cambiar el nivel de registro a INFO, puede usar `containerProperties.setCommitLogLevel ( LogIfLevelEnabled.Level.INFO) ;`.

A partir de la versión 2.2, se ha agregado una nueva propiedad de contenedor llamada `missingTopicsFatal` (valor predeterminado: verdadero). Si no hay temas configurados en el agente, se bloqueará el inicio del contenedor. No aplicable si el contenedor está configurado para escuchar en el modo de tema (expresión regular). Anteriormente, los subprocesos de contenedor se conectaban a través del método `consumer.poll ()`, esperando mostrar el asunto cuando se registraban muchos mensajes. No hay indicación de un problema que no sea el registro. Para restaurar el comportamiento anterior, puede establecer esta propiedad en falso.


### 3.2.2 Uso de `ConcurrentMessageListenerContainer`

Un solo constructor es similar al primer constructor `KafkaListenerContainer`. La siguiente lista muestra la firma del constructor:

```

1 | public ConcurrentMessageListenerContainer(ConsumerFactory<K, V> consumerFactory,
2 |     ContainerProperties containerProperties)

```

También tiene propiedades de concurrencia. Por ejemplo, `container.setConcurrency (3)` crea tres instancias `KafkaMessageListenerContainer`. 

Para el primer constructor, Kafka utiliza sus capacidades de gestión de grupo para asignar particiones entre los consumidores.

Al escuchar varios temas, la distribución de partición predeterminada puede ser diferente de lo que cabría esperar. Por ejemplo, si tiene tres temas, cada tema tiene cinco particiones y desea usar concurrencia =

15, solo puede ver cinco usuarios activos, cada usuario principal tiene asignada una partición y los otros 10 usuarios están inactivos. Esto se debe a que el Kafka

PartitionAssignor predeterminado es RangeAssignor (consulte su Javadoc).

Para este caso, es posible que desee considerar el uso de RoundRobinAssignor, que asigna la partición a todos los usuarios. Luego, asigne un tema o partición a cada usuario.

Para cambiar el PartitionAssignor, establezca la propiedad del consumidor `Partition.assignment.strategy` (`ConsumerConfigs.PARTITION_ASSIGNMENT_STRATEGY_CONFIG`) en las propiedades proporcionadas a `DefaultKafkaConsumerFactory`.

Al usar Spring Boot, puede asignar políticas de la siguiente manera:

```
1 | spring.kafka.consumer.properties.partition.assignment.strategy=\
2 | org.apache.kafka.clients.consumer.RoundRobinAssignor
```

Para el segundo constructor, `ConcurrentMessageListenerContainer` distribuye la instancia `TopicPartition` en la instancia delegada `KafkaMessageListenerContainer`.

Por ejemplo, si se proporcionan seis instancias de `TopicPartition` y la concurrencia es 3; Cada contenedor tiene dos particiones. Para cinco instancias de `TopicPartition`, dos contenedores obtienen dos particiones y el tercero obtiene una. Si la concurrencia es mayor que la cantidad de `TopicPartitions`, ajuste la concurrencia para obtener una partición por contenedor.

El atributo `client.id` (si está configurado) agrega `-n`, donde `n` es la instancia del consumidor correspondiente a la concurrencia. Cuando habilita JMX, debe darle al MBean un nombre único.

A partir de la versión 1.3, `MessageListenerContainer` proporciona acceso a las métricas subyacentes de `KafkaConsumer`. Para `ConcurrentMessageListenerContainer`, el método `metrics()` devuelve las métricas para todas las instancias de `KafkaMessageListenerContainer` de destino. Indicadores de agrupación para asignar `<MetricName ,? Extienda Métrica>` por el ID de cliente proporcionado al `KafkaConsumer` subyacente.

### 3.3 Compromiso de compensaciones Envío de compensaciones

Se proporcionan varias opciones para enviar compensaciones. Si la propiedad del consumidor `enable.auto.commit` es verdadera, Kafka enviará automáticamente la compensación en función

de su configuración. Si es falso, el contenedor admite múltiples configuraciones de AckMode (descritas en la siguiente lista).

El método de encuesta al consumidor () devuelve uno o más ConsumerRecords. Llame a un MessageListener para cada registro. La siguiente lista describe las acciones tomadas por el contenedor para cada AckMode:

- REGISTRO: El desplazamiento se envía cuando el oyente regresa después de procesar el registro.
- LOTE: envía un desplazamiento al procesar todos los registros devueltos por poll ().
- TIME: el desplazamiento en el que se procesan todos los registros devueltos por poll (), siempre que se haya excedido el tiempo de confirmación desde la última confirmación.
- COUNT: siempre que se haya recibido el registro ackCount desde la última confirmación, el desplazamiento se envía cuando se procesan todos los registros devueltos por poll ().
- COUNT\_TIME: similar a TIME y COUNT, pero si alguna de las condiciones es verdadera, se realiza la confirmación.
- MANUAL: El oyente del mensaje es responsable de confirmar () la confirmación. Después de eso, aplique la misma semántica que BATCH.
- MANUAL\_IMMEDIATE: envíe el desplazamiento inmediatamente cuando el oyente llame al método Acknowledgment.acknowledge ().

MANUAL y MANUAL\_IMMEDIATE requieren que el oyente sea un AcknowledgingMessageListener o un BatchAcknowledgingMessageListener.

Ver el oyente del mensaje.

Use el método commitSync () o commitAsync () en el consumidor según la propiedad del contenedor syncCommits.

Confirme que tiene los siguientes métodos:

```
1 public interface Acknowledgment {  
2  
3     void acknowledge();  
4  
5 }
```

Este método permite al oyente controlar cuándo se confirma un desplazamiento.

Intimidación

## Inicio automático del contenedor de escucha

El contenedor de escucha implementa SmartLifecycle, que por defecto es autoStartup. El contenedor se inicia más tarde (Integer.MAX-VALUE - 100). Otros componentes que implementan SmartLifecycle para procesar datos del oyente deben iniciarse en una etapa temprana. -100 deja espacio para futuras fases, permitiendo que los componentes se inicien automáticamente después del contenedor.

## Anotación @KafkaListener

La anotación @KafkaListener se utiliza para especificar el método de bean como escucha para el contenedor de escucha. El bean está contenido en MessagingMessageListenerAdapter, que está configurado con varias funciones, como un convertidor para convertir datos para que coincidan con los parámetros del método cuando sea necesario.

Puede usar SpEL para configurar la mayoría de las propiedades en anotaciones usando # {...} o marcadores de posición de atributo (\$ {...}). Vea el Javadoc para más información.

## Grabar oyentes

La anotación @KafkaListener proporciona un mecanismo para oyentes POJO simples. El siguiente ejemplo muestra cómo usarlo:

```
1      public class Listener {  
2  
3          @KafkaListener(id = "foo", topics = "myTopic", clientIdPrefix = "myClientId")  
4              public void listen(String data) {  
5                  ...  
6              }  
7  
8          }
```

Este mecanismo requiere el uso de anotaciones @EnableKafka en una de las clases de @Configuration y una fábrica de contenedores de escucha, que se utiliza para configurar el ConcurrentMessageListenerContainer subyacente. Por defecto, se requiere un bean llamado kafkaListenerContainerFactory. El siguiente ejemplo muestra cómo usar ConcurrentMessageListenerContainer:

```
1      @Configuration  
2      @EnableKafka  
3      public class KafkaConfig {  
4  
5          @Bean  
6          kafkaListenerContainerFactory<ConcurrentMessageListenerContainer<Ir  
7              kafkaListenerContainerFactory() {
```



```

8      ConcurrentKafkaListenerContainerFactory<Integer, String> factory =
9          new ConcurrentKafkaListenerContainerFactory<>();
10         factory.setConsumerFactory(consumerFactory());
11         factory.setConcurrency(3);
12         factory.getContainerProperties().setPollTimeout(3000);
13         return factory;
14     }
15
16     @Bean
17     public ConsumerFactory<Integer, String> consumerFactory() {
18         return new DefaultKafkaConsumerFactory<>(consumerConfigs());
19     }
20
21     @Bean
22     public Map<String, Object> consumerConfigs() {
23         Map<String, Object> props = new HashMap<>();
24         props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, embeddedKafka.getBroker
25             ...
26         return props;
27     }
28     }

```

Tenga en cuenta que para establecer las propiedades del contenedor, debe usar el método `getContainerProperties()` en su fábrica. Se utiliza como plantilla para inyectar las propiedades reales del contenedor.

A partir de la versión 2.1.1, ahora puede establecer la propiedad `client.id` para el consumidor que crea la anotación. `ClientIdPrefix` tiene el sufijo `-n`, donde `n` es un número entero que indica el número del contenedor cuando se usa la concurrencia.

A partir de la versión 2.2, ahora puede anular las propiedades de simultaneidad y `AutoStartup` de la fábrica de contenedores utilizando las propiedades de la propia anotación. Los atributos pueden ser valores simples, marcadores de posición de atributos o expresiones SpEL. El siguiente ejemplo muestra cómo hacer esto:

```

1      @KafkaListener(id = "myListener", topics = "myTopic",
2      autoStartup = "${listen.auto.start:true}", concurrency = "${listen.concur
3      public void listen(String data) {
4          ...
5      }

```

También puede configurar oyentes POJO con temas explícitos y particiones (y compensaciones iniciales opcionales). El siguiente ejemplo muestra cómo hacer esto:

Intimidación

```

1      @KafkaListener(id = "thing2", topicPartitions =
2      { @TopicPartition(topic = "topic1", partitions = { "0", "1" }),
3        @TopicPartition(topic = "topic2", partitions = "0",
4        partitionOffsets = @PartitionOffset(partition = "1", initialOffset =
5                               })
6        public void listen(ConsumerRecord<?, ?> record) {
7            ...
8        }

```

Puede especificar cada partición en la partición o en el atributo divisionOffsets, pero no en ambos.

También puede proporcionar confirmación al oyente cuando utiliza AckMode manual. El siguiente ejemplo también muestra cómo usar otras fábricas de contenedores.

```

1      @KafkaListener(id = "cat", topics = "myTopic",
2      containerFactory = "kafkaManualAckListenerContainerFactory")
3      public void listen(String data, Acknowledgment ack) {
4          ...
5          ack.acknowledge();
6      }

```

Finalmente, puede obtener metadatos sobre el mensaje del encabezado. Puede usar los siguientes nombres de encabezado para recuperar el encabezado de un mensaje:

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

El siguiente ejemplo muestra cómo usar el encabezado:

```

1      @KafkaListener(id = "qux", topicPattern = "myTopic1")
2      public void listen(@Payload String foo,
3      @Header(KafkaHeaders.RECEIVED_MESSAGE_KEY) Integer key,
4      @Header(KafkaHeaders.RECEIVED_PARTITION_ID) int partition,
5      @Header(KafkaHeaders.RECEIVED_TOPIC) String topic,
6      @Header(KafkaHeaders.RECEIVED_TIMESTAMP) long ts
7          ) {
8          ...

```



## Oyentes por lotes

A partir de la versión 1.1, puede configurar el método `@KafkaListener` para recibir todo el lote de registros de consumidores recibidos de encuestas de consumidores. Para configurar la fábrica de contenedores de escucha para crear un escucha por lotes, puede establecer la propiedad `batchListener`. El siguiente ejemplo muestra cómo hacer esto:

```
1         @Bean  
2         public KafkaListenerContainerFactory<?> batchFactory() {  
3             ConcurrentKafkaListenerContainerFactory<Integer, String> factory =  
4                 new ConcurrentKafkaListenerContainerFactory<>();  
5                 factory.setConsumerFactory(consumerFactory());  
6                 factory.setBatchListener(true); // <<<<<<<<<<<<<<<<<<<<  
7                 return factory;  
8             }
```

El siguiente ejemplo muestra cómo recibir una lista de cargas útiles

```
1 @KafkaListener(id = "list", topics = "myTopic", containerFactory = "batchFactory"
2     public void listen(List<String> list) {
3         ...
4     }
```

Los temas, particiones, compensaciones, etc. están disponibles en el encabezado paralelo a la carga útil. El siguiente ejemplo muestra cómo usar el encabezado:

```
1 @KafkaListener(id = "list", topics = "myTopic", containerFactory = "batchFactory")
2     public void listen(List<String> list,
3         @Header(KafkaHeaders.RECEIVED_MESSAGE_KEY) List<Integer> keys,
4         @Header(KafkaHeaders.RECEIVED_PARTITION_ID) List<Integer> partitions,
5         @Header(KafkaHeaders.RECEIVED_TOPIC) List<String> topics,
6         @Header(KafkaHeaders.OFFSET) List<Long> offsets) {
7         ...
8     }
```

¿O puede recibir un mensaje con cada desplazamiento y otros detalles en cada mensaje <? > La lista de objetos, pero debe ser un parámetro único (a excepción de la confirmación opcional, cuando se utiliza la confirmación manual, y / o los parámetros del consumidor <?,?>) Están definidos en el método. El siguiente ejemplo muestra cómo hacer esto:

```
1 @KafkaListener(id = "listMsg", topics = "myTopic", containerFactory = "hatchFacto
2     public void listen14(List<Message<?>> list) {
```

```

3          ...
4      }
5
6  @KafkaListener(id = "listMsgAck", topics = "myTopic", containerFactory = "batchFa
7      public void listen15(List<Message<?>> list, Acknowledgment ack) {
8          ...
9      }
10
11 @KafkaListener(id = "listMsgAckConsumer", topics = "myTopic", containerFactory =
12 public void listen16(List<Message<?>> list, Acknowledgment ack, Consumer<?, ?> co
13     ...
14     }

```

En este caso, no se realiza ninguna conversión en la carga útil.

Si `BatchMessagingMessageConverter` está configurado con un `RecordMessageConverter`, también puede agregar un tipo genérico al parámetro Mensaje y convertir la carga útil. Para obtener más información, consulte [Conversión de carga útil con escuchas masivas](#)

También puede recibir `ConsumerRecord <?, ?>` Una lista de objetos, pero debe ser el único parámetro definido en el método (a excepción de la confirmación opcional cuando se utiliza la confirmación manual y los parámetros `Consumer <?,?>`). El siguiente ejemplo muestra cómo hacer esto:

```

1  @KafkaListener(id = "listCRs", topics = "myTopic", containerFactory = "batchFacto
2      public void listen(List<ConsumerRecord<Integer, String>> list) {
3          ...
4      }
5
6  @KafkaListener(id = "listCRsAck", topics = "myTopic", containerFactory = "batchFa
7  public void listen(List<ConsumerRecord<Integer, String>> list, Acknowledgment ack
8      ...
9      }

```

A partir de la versión 2.2, el oyente puede recibir los `ConsumerRecords` completos devueltos por el método `poll ()`. `?, ?>` Objeto, permita que el oyente acceda a otros métodos, tales como `particiones ()` (devolviendo instancias de `TopicPartition` en la lista) y `registros (TopicPartition)` (obteniendo registros selectivos). Nuevamente, este debe ser el único parámetro en el método (a excepción de la confirmación opcional cuando se utilizan los parámetros de confirmación manual o de consumidor `<?,?>`). El siguiente ejemplo muestra cómo hacer esto:

```

1  @KafkaListener(id = "pollResults", topics = "myTopic", containerFactory
2      public void pollResults(ConsumerRecords<?, ?> records) {

```

Intimidación

```

3      ...
4      }

```

ConsumerRecords se ignora si la fábrica de contenedores está configurada con RecordFilterStrategy <? ,? > Escucha y emite un mensaje de registro WARN. Si usa <Lista <? >> Escuchas formales que solo usan escuchas masivas para filtrar registros.

## Atributo de anotación

A partir de la versión 2.0, el atributo id (si existe) se usa como el atributo de usuario group.id de Kafka, anulando los atributos configurados en la fábrica del consumidor, si existen. También puede establecer explícitamente groupId o establecer idIsGroup en false para reanudar el comportamiento anterior de usar el consumidor factory group.id.

Puede usar marcadores de posición de atributos o expresiones SpEL en la mayoría de las propiedades de anotación, como se muestra en el siguiente ejemplo:

```

1      @KafkaListener(topics = "${some.property}")
2
3      @KafkaListener(topics = "#{someBean.someProperty}",
4                     groupId = "#{someBean.someProperty}.group")

```

A partir de la versión 2.1.2, las expresiones SpEL admiten un token especial: \_\_listener. Es un nombre de pseudo bean que representa la instancia de bean actual para esta anotación.

Considere el siguiente ejemplo:

```

1      @Bean
2      public Listener listener1() {
3          return new Listener("topic1");
4      }
5
6      @Bean
7      public Listener listener2() {
8          return new Listener("topic2");
9      }

```

Dados los beans en el ejemplo anterior, podemos usar lo siguiente:

```

1      public class Listener {
2
3          private final String topic;

```



```

4
5         public Listener(String topic) {
6             this.topic = topic;
7         }
8
9         @KafkaListener(topics = "#{__listener.topic}",
10            groupId = "#{__listener.topic}.group")
11            public void listen(...) {
12                ...
13            }
14
15            public String getTopic() {
16                return this.topic;
17            }
18
19        }

```

Si tiene un bean real llamado `__listener`, puede cambiar la etiqueta de expresión usando la propiedad `beanRef`. El siguiente ejemplo muestra cómo hacer esto:

```

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

```

A partir de la versión 2.2.4, puede especificar las propiedades del usuario de Kafka directamente en las anotaciones que anulan cualquier propiedad con el mismo nombre configurado en la fábrica del consumidor. No puede especificar las propiedades `group.id` y `client.id` de esta manera; serán ignorados; use las propiedades de anotación `groupId` y `clientIdPrefix`.

El atributo se especifica como una sola cadena con el formato de archivo de propiedades Java normal: `foo: bar`, `foo = bar` o `foo bar`.

```

1         @KafkaListener(topics = "myTopic", groupId="group", properties= {
2             "max.poll.interval.ms:60000",
3             ConsumerConfig.MAX_POLL_RECORDS_CONFIG + "=100"
4         })

```

## Nombre del hilo del contenedor

El contenedor de escucha actualmente utiliza dos ejecutores de tareas, uno para llamar al consumidor y el otro para llamar al escucha cuando la propiedad del consumidor `kafka.enable.auto.commit` es falsa. Puede proporcionar un ejecutable personalizado estableciendo las propiedades `consumerExecutor` y `listenerExecutor` de `Container's ContainerProperties`. Cuando use ejecutores agrupados, asegúrese de que haya suficientes subprocesos disponibles para



manejar la concurrencia de todos los contenedores que los usan. Cuando se usa `ConcurrentMessageListenerContainer`, cada consumidor usa un hilo (concurrente).

Si no proporciona un usuario para ejecutar el programa, use `SimpleAsyncTaskExecutor`. Este ejecutor crea un hilo con un nombre similar a `-C-1` (el hilo del consumidor). Para `ConcurrentMessageListenerContainer`, la parte del nombre del subproceso se convierte en `-m`, donde `m` representa la instancia del consumidor. `n` aumenta cada vez que se inicia el contenedor. Por lo tanto, utilizando el nombre del bean del contenedor, después de que el contenedor se inicie por primera vez, los hilos en este contenedor se denominarán `container-0-C-1`, `container-1-C-1`, etc.; `container-0-C-2`, `container-1-C-2`, etc., deténgase y luego comience.

## @KafkaListener como una meta anotación

A partir de la versión 2.2, ahora puede usar `@KafkaListener` como una metaanotación. El siguiente ejemplo muestra cómo hacer esto:

```

1      @Target(ElementType.METHOD)
2      @Retention(RetentionPolicy.RUNTIME)
3      @KafkaListener
4      public @interface MyThreeConsumersListener {
5
6          @AliasFor(annotation = KafkaListener.class, attribute = "id")
7              String id();
8
9          @AliasFor(annotation = KafkaListener.class, attribute = "topics")
10             String[] topics();
11
12         @AliasFor(annotation = KafkaListener.class, attribute = "concurrency")
13             String concurrency() default "3";
14
15     }
```

A menos que se especifique `group.id` en la configuración de fábrica del consumidor. De lo contrario, debe agregar un alias a al menos uno de los temas, `topicPattern` o `topicPartitions` (y generalmente `id` o `groupId`). El siguiente ejemplo muestra cómo hacer esto:

```

1      @MyThreeConsumersListener(id = "my.group", topics = "my.topic")
2      public void listen1(String in) {
3          ...
4      }
```

## Usa @KafkaListener en una clase



Cuando utilice `@KafkaListener` en el nivel de clase, debe especificar `@KafkaHandler` en el nivel de método. Cuando se entrega un mensaje, el tipo de carga útil del mensaje convertido se usa para determinar el método a llamar. El siguiente ejemplo muestra cómo hacer esto:

```
1      @KafkaListener(id = "multi", topics = "myTopic")
2          static class MultiListenerBean {
3
4              @KafkaHandler
5              public void listen(String foo) {
6                  ...
7              }
8
9              @KafkaHandler
10             public void listen(Integer bar) {
11                 ...
12             }
13
14             @KafkaHandler(isDefault = true)
15             public void listenDefault(Object object) {
16                 ...
17             }
18
19         }
```

A partir de la versión 2.1.3, si no coincide con otros métodos, puede especificar el método `@KafkaHandler` como el método predeterminado para llamar. Puede especificar hasta un método. Cuando se utiliza el método `@KafkaHandler`, la carga útil debe haberse convertido en un objeto de dominio (para que se pueda realizar una coincidencia). Utilice un deserializador personalizado, `JsonDeserializer` o `(String | Bytes) JsonMessageConverter` y establezca su `TypePrecedence` en `TYPE_ID`. Para obtener más información, consulte [Serialización, Deserialización y Transformación de mensajes](#).

## `@KafkaListener` Gestión del ciclo de vida

El contenedor de escucha creado para la anotación `@KafkaListener` no es un bean en el contexto de la aplicación. En cambio, están registrados con un bean de infraestructura del tipo `KafkaListenerEndpointRegistry`. El bean declara y administra este bean automáticamente para administrar el ciclo de vida del contenedor; iniciará automáticamente cualquier contenedor con `autoStartup` establecido en `verdadero`. Todos los contenedores creados por todas las fábricas de contenedores deben estar en la misma etapa. Para obtener más información, consulte [Inicio automático del contenedor de escucha](#). Puede usar el registro para administrar programáticamente el ciclo de vida. Iniciar o detener el registro iniciará o detendrá todos los contenedores registrados. Alternativamente, puede usar su atributo `id` para obtener una referencia a un solo contenedor. Puede configurar `autoStartup` en la anotación, [Intimidación](#)



configuración predeterminada configurada en la fábrica de contenedores. Puede obtener una referencia a un bean desde el contexto de la aplicación, como el enrutamiento automático, para administrar sus contenedores registrados. El siguiente ejemplo muestra cómo hacer esto:

```

1  @KafkaListener(id = "myContainer", topics = "myTopic", autoStartup = "false")
2      public void listen(...) { ... }

1      @Autowired
2      private KafkaListenerEndpointRegistry registry;
3
4      ...
5
6      this.registry.getListenerContainer("myContainer").start();
7
8      ...

```

## @KafkaListener @Payload check

A partir de la versión 2.2, ahora es más fácil agregar un Validador para validar el parámetro @KafkaListener @Payload. Anteriormente, tenía que configurar un DefaultMessageHandlerMethodFactory personalizado y agregarlo al registrador. Ahora puede agregar el validador al registrador mismo. El siguiente código muestra cómo hacer esto:

```

1      @Configuration
2      @EnableKafka
3      public class Config implements KafkaListenerConfigurer {
4
5          ...
6
7          @Override
8          public void configureKafkaListeners(KafkaListenerEndpointRegistrar registrar) {
9              registrar.setValidator(new MyValidator());
10             }
11         }

```

Cuando se usa Spring Boot con Validation Launcher, LocalValidatorFactoryBean se configura automáticamente, como se muestra en el siguiente ejemplo:

```

1      @Configuration
2      @EnableKafka
3      public class Config implements KafkaListenerConfigurer {
4
5          @Autowired

```



```

6         private LocalValidatorFactoryBean validator;
7
8
9         @Override
10        public void configureKafkaListeners(KafkaListenerEndpointRegistrar registrar)
11            registrar.setValidator(this.validator);
12        }
13    }

```

El siguiente ejemplo muestra cómo verificar:

```

1        public static class ValidatedClass {
2
3            @Max(10)
4            private int bar;
5
6            public int getBar() {
7                return this.bar;
8            }
9
10           public void setBar(int bar) {
11               this.bar = bar;
12           }
13
14       }

```

```

1  @KafkaListener(id="validated", topics = "annotated35", errorHandler = "validation
2      containerFactory = "kafkaJsonListenerContainerFactory")
3      public void validatedListener(@Payload @Valid ValidatedClass val) {
4          ...
5      }
6
7      @Bean
8      public KafkaListenerErrorHandler validationErrorHandler() {
9          return (m, e) -> {
10              ...
11          };
12      }

```

## Rebalanceo de oyentes



ContainerProperties tiene una propiedad llamada consumerRebalanceListener que acepta la implementación de la interfaz ConsumerRebalanceListener del cliente Kafka.

Intimidación

proporciona esta propiedad, el contenedor configurará una escucha de grabación para grabar eventos de reequilibrio en el nivel INFO. El marco también agrega una subinterfaz `ConsumerAwareRebalanceListener`. La siguiente lista muestra la definición de interfaz `ConsumerAwareRebalanceListener`:

```
1 public interface ConsumerAwareRebalanceListener extends ConsumerRebalanceListener
2
3     void onPartitionsRevokedBeforeCommit(Consumer<?, ?> consumer, Collection<Topic
4
5     void onPartitionsRevokedAfterCommit(Consumer<?, ?> consumer, Collection<Topic
6
7     void onPartitionsAssigned(Consumer<?, ?> consumer, Collection<TopicPartition>
8
9     }
```

Tenga en cuenta que hay dos devoluciones de llamada al deshacer una partición. El primero se llama de inmediato. Llame al segundo después de cometer cualquier compensación pendiente. Esto es útil si desea mantener compensaciones en algunos repositorios externos, como se muestra en el siguiente ejemplo:

```
1 containerProperties.setConsumerRebalanceListener(new ConsumerAwareRebalanceListen
2
3     @Override
4     public void onPartitionsRevokedBeforeCommit(Consumer<?, ?> consumer, Collecti
5         // acknowledge any pending Acknowledgments (if using manual acks)
6     }
7
8     @Override
9     public void onPartitionsRevokedAfterCommit(Consumer<?, ?> consumer, Collectio
10         // ...
11         store(consumer.position(partition));
12         // ...
13     }
14
15     @Override
16     public void onPartitionsAssigned(Collection<TopicPartition> partitions) {
17         // ...
18         consumer.seek(partition, offsetTracker.getOffset() + 1);
19         // ...
20     }
21     });
```



A partir de la versión 2.0, si también utiliza el comentario de anotación `@SendTo` `@KafkaListener` y la llamada al método devuelve el resultado, el resultado se reenviará al tema especificado por `@SendTo`.

El valor `@SendTo` puede tomar muchas formas:

- `@SendTo ("someTopic")` rutas a temas de texto
- `@SendTo ("# {someExpression}")` enruta al sujeto determinado por la expresión de cálculo durante la inicialización del contexto de la aplicación.
- `@SendTo ("! {SomeExpression}")` se enruta al sujeto determinado por la expresión en tiempo de ejecución. El objeto `#root` evaluado tiene tres propiedades:
  - Solicitud: `Inbound ConsumerRecord` (o el objeto `ConsumerRecords` del oyente por lotes))
  - Fuente: `org.springframework.messaging.Message` <? convertido de solicitud>.
  - Resultado: el método devuelve el resultado.
- `@SendTo` (sin atributo): ¡Esto se considera! `{source.headers ['kafka_replyTopic']}` (desde la versión 2.1.3).

A partir de las versiones 2.1.11 y 2.2.1, los marcadores de posición de atributo se resuelven dentro del valor `@SendTo`.

El resultado de la evaluación de la expresión debe ser una Cadena que represente el nombre del sujeto. El siguiente ejemplo muestra varias formas de usar `@SendTo`:

```

1      @KafkaListener(topics = "annotated21")
2      @SendTo("!{request.value()}") // runtime SpEL
3      public String replyingListener(String in) {
4          ...
5      }
6
7      @KafkaListener(topics = "${some.property:annotated22}")
8      @SendTo("#{myBean.replyTopic}") // config time SpEL
9      public Collection<String> replyingBatchListener(List<String> in) {
10         ...
11     }
12
13     @KafkaListener(topics = "annotated23", errorHandler = "replyErrorHandler")
14     @SendTo("annotated23reply") // static reply topic definition
15     public String replyingListenerWithErrorHandler(String in) {

```

```

16         ...
17     }
18     ...
19     @KafkaListener(topics = "annotated25")
20     @SendTo("annotated25reply1")
21     public class MultiListenerSendTo {
22
23         @KafkaHandler
24         public String foo(String in) {
25             ...
26         }
27
28         @KafkaHandler
29         @SendTo("!{'annotated25reply2'}")
30         public String bar(@Payload(required = false) KafkaNull nul,
31             @Header(KafkaHeaders.RECEIVED_MESSAGE_KEY) int key) {
32             ...
33         }
34
35     }

```

A partir de la versión 2.2, puede agregar ReplyHeadersConfigurer a la fábrica de contenedores de escucha. Revise esta información para determinar qué encabezados establecer en el mensaje de respuesta. El siguiente ejemplo muestra cómo agregar un ReplyHeadersConfigurer:

```

1         @Bean
2     public ConcurrentKafkaListenerContainerFactory<Integer, String> kafkaListenerCont
3         ConcurrentKafkaListenerContainerFactory<Integer, String> factory =
4         new ConcurrentKafkaListenerContainerFactory<>();
5         factory.setConsumerFactory(cf());
6         factory.setReplyTemplate(template());
7         factory.setReplyHeadersConfigurer((k, v) -> k.equals("cat"));
8         return factory;
9     }

```

También puede agregar más títulos si lo desea. El siguiente ejemplo muestra cómo hacer esto:

```

1         @Bean
2     public ConcurrentKafkaListenerContainerFactory<Integer, String> kafkaListenerCont
3         ConcurrentKafkaListenerContainerFactory<Integer, String> factory =
4         new ConcurrentKafkaListenerContainerFactory<>();
5         factory.setConsumerFactory(cf());
6         factory.setReplyTemplate(template());
7         factory.setReplyHeadersConfigurer(new ReplyHeadersConfigurer() {
8

```



```

9         @Override
10        public boolean shouldCopy(String headerName, Object headerValue) {
11            return false;
12        }
13
14        @Override
15        public Map<String, Object> additionalHeaders() {
16            return Collections.singletonMap("qux", "fiz");
17        }
18
19        });
20        return factory;
21    }

```

Cuando use `@SendTo`, debe usar `KafkaTemplate` para configurar `ConcurrentKafkaListenerContainerFactory` en su propiedad `replyTemplate` para realizar el envío.

A menos que use la semántica de solicitud / respuesta, solo usa el método simple de envío (tema, valor), por lo que es posible que desee crear una subclase para generar particiones o claves.

El siguiente ejemplo muestra cómo hacer esto:

```

1         @Bean
2        public KafkaTemplate<String, String> myReplyingTemplate() {
3            return new KafkaTemplate<Integer, String>(producerFactory()) {
4
5                @Override
6                public ListenableFuture<SendResult<String, String>> send(String topic, St
7                    return super.send(topic, partitionForData(data), keyForData(data), da
8                }
9
10                ...
11
12            };
13        }

```

Si el método de escucha devuelve un mensaje `<? >` o Colección `<Mensaje <? >>`, el método de escucha es responsable de configurar el encabezado de respuesta. Por ejemplo, cuando procesa una solicitud de `ReplyKafkaTemplate`, puede hacer lo siguiente:

```

1        @KafkaListener(id = "messageReturned", topics = "someTopic")
2        public Message<?> listen(String in, @Header(KafkaHeaders.REPLY_TOPIC) byte[] repl
3            @Header(KafkaHeaders.CORRELATION_ID) byte[] correlation

```

Intimidación

```

4         return MessageBuilder.withPayload(in.toUpperCase())
5             .setHeader(KafkaHeaders.TOPIC, replyTo)
6             .setHeader(KafkaHeaders.MESSAGE_KEY, 42)
7             .setHeader(KafkaHeaders.CORRELATION_ID, correlation)
8             .setHeader("someOtherHeader", "someValue")
9             .build();
10    }

```

Al utilizar la semántica de solicitud / respuesta, el remitente puede solicitar la partición de destino.

Incluso si no se devuelven resultados, puede usar `@SendTo` para anotar el método `@KafkaListener`. Esto es para permitir que se configure el `errorHandler`, que reenvía información sobre mensajes fallidos a un tema. El siguiente ejemplo muestra cómo hacer esto:

```

1  KafkaListener(id = "voidListenerWithReplyingErrorHandler", topics = "someTopic",
2              errorHandler = "voidSendToErrorHandler")
3              @SendTo("failures")
4  public void voidListenerWithReplyingErrorHandler(String in) {
5      throw new RuntimeException("fail");
6  }
7
8      @Bean
9  public KafkaListenerErrorHandler voidSendToErrorHandler() {
10      return (m, e) -> {
11          return ... // some information about the failure and input data
12      };
13  }

```

Para obtener más información, consulte Manejo de excepciones.

## Filtrando mensajes

En algunos casos, como el reequilibrio, los mensajes que ya se han procesado se pueden volver a entregar. El marco no puede saber si dicho mensaje ha sido procesado. Esta es una característica de nivel de aplicación. Esto se denomina patrón de receptor idempotente, y Spring Integration proporciona su implementación.

El proyecto Spring for Apache Kafka también proporciona ayuda a través de la clase `FilteringMessageListenerAdapter`, que envuelve `MessageListener`. Esta clase utiliza la implementación de `RecordFilterStrategy`, donde puede implementar el método de filtro para indicar que el mensaje está duplicado y debe descartarse. Esto tiene una propie

Intimidación

llamada `ackDiscarded` que indica si el adaptador debe reconocer el registro descartado. Por defecto es falso.

Cuando use `@KafkaListener`, configure `RecordFilterStrategy` (y opcionalmente `ackDiscarded`) en la fábrica de contenedores para envolver el oyente en el adaptador de filtro apropiado.

Además, se proporciona `FilteringBatchMessageListenerAdapter` para usar cuando se utilizan escuchas de mensajes por lotes.

Si `@KafkaListener` recibe `ConsumerRecords` `<?, ? >` en lugar de `List <ConsumerRecord <?, ? >>`, ignore `FilteringBatchMessageListenerAdapter` porque `ConsumerRecords` es inmutable.

## Reintentando entregas

Si el oyente lanza una excepción, el comportamiento predeterminado es llamar a `ErrorHandler` (si está configurado) o de otra manera.

Se proporcionan dos interfaces de controlador de errores (`ErrorHandler` y `BatchErrorHandler`). Debe configurar el tipo apropiado para que coincida con el escucha de mensajes.

Para volver a intentar la entrega, se proporciona un conveniente adaptador de escucha `RetryingMessageListenerAdapter`.

Puede configurarlo con `RetryTemplate` y `RecoveryCallback`; consulte el proyecto `spring-retry` para obtener información sobre estos componentes. Si no se proporciona una devolución de llamada de recuperación, se lanzará una excepción al contenedor después de que se agote el reintento. En este caso, si está configurado, se llama a `ErrorHandler`; de lo contrario, se registrará.

Al usar `@KafkaListener`, puede configurar `RetryTemplate` (y `recoveryCallback` opcional) en la fábrica de contenedores. Cuando hace esto, el oyente está envuelto en el adaptador de reintento apropiado.

El contenido del `RetryContext` pasado a `RecoveryCallback` depende del tipo de escucha. El contexto siempre tiene un atributo de registro, que es un registro de la falla. Si su oyente está confirmando o es conocido por el consumidor, puede usar otros atributos de confirmación o de usuario. Por conveniencia, `RetryingMessageListenerAdapter` proporciona constantes estáticas para estas claves. Vea su Javadoc para más información.

El adaptador de reintento no se proporciona para los escuchas de mensajes por lotes porque el marco no sabe dónde se produjo el error en el lote. Si necesita volver a intentar las características cuando usa un escucha masivo, le recomendamos que use `RetryTemplate` en el mismo escucha.



## Reintento con estado

Debe tener en cuenta que el reintento discutido en la sección anterior suspenderá el hilo del consumidor (si usa `BackOffPolicy`). No se llamó a `Consumer.poll ()` durante el reintento. Kafka tiene dos atributos para determinar la salud del consumidor. `Session.timeout.ms` se usa para determinar si el usuario está activo. A partir de la versión 0.10.1.0, el latido se envía en un hilo de fondo, por lo que los consumidores lentos ya no lo afectan. `max.poll.interval.ms` (Predeterminado: cinco minutos) Se usa para determinar si el consumidor parece estar suspendido (tardó demasiado en procesar el registro de la última encuesta). Si el tiempo entre las llamadas `poll ()` excede este valor, el agente deshará la partición asignada y realizará un reequilibrio. Para secuencias de reintento largas, esto puede suceder fácilmente al retirarse.

A partir de la versión 2.1.3, puede evitar este problema utilizando el reintento de estado con `SeekToCurrentErrorHandler`. En este caso, cada intento de pase devuelve la excepción al contenedor, el controlador de errores vuelve a buscar el desplazamiento no controlado y la próxima encuesta `()` retransmitirá el mismo mensaje. Esto evita problemas más allá de la propiedad `max.poll.interval.ms` (siempre que el retraso único entre intentos no la exceda). Por lo tanto, cuando use `ExponentialBackOffPolicy`, debe asegurarse de que `maxInterval` sea menor que la propiedad `max.poll.interval.ms`. Para habilitar el reintento con estado, puede usar el constructor `RetryingMessageListenerAdapter` con un parámetro booleano de estado (configúrelo como verdadero). Al configurar la fábrica de contenedores de escucha (para `@KafkaListener`), establezca la propiedad `statefulRetry` de la fábrica en `true`.

## Detección de consumidores inactivos y que no responden

Si bien es efectivo, un problema para los consumidores asíncronos es detectar cuándo están inactivos. Si no llega ningún mensaje por un tiempo, es posible que deba tomar alguna medida.

Puede configurar el contenedor de escucha para publicar un `ListenerContainerIdleEvent` después de un período de tiempo sin mensajes. Cuando el contenedor está inactivo, se publica un evento cada milisegundo `idleEventInterval`.

Para configurar esta función, establezca `idleEventInterval` en el contenedor. El siguiente ejemplo muestra cómo hacer esto:

```
1      @Bean
2  public KafkaMessageListenerContainer(ConsumerFactory<String, String> consumerFact
3      ContainerProperties containerProps = new ContainerProperties("topic1", "topic
4      ...
5      containerProps.setIdleEventInterval(60000L);
6      ...
7      KafkaMessageListenerContainer<String, String> container = new KafKaMessageLis
8      return container;
9      }
```

 Intimididad

El siguiente ejemplo muestra cómo configurar `idleEventInterval` para `@KafkaListener`:

```
1      @Bean
2      public ConcurrentKafkaListenerContainerFactory kafkaListenerContainerFactory() {
3          ConcurrentKafkaListenerContainerFactory<String, String> factory =
4              new ConcurrentKafkaListenerContainerFactory<>();
5              ...
6          factory.getContainerProperties().setIdleEventInterval(60000L);
7          ...
8          return factory;
9      }
```

En cada caso, el evento se publica cada minuto cuando el contenedor está inactivo.

Además, si no se puede acceder al agente, el método de encuesta al consumidor () no se cierra, por lo que no se reciben mensajes ni se generan eventos inactivos. Para resolver este problema, si la encuesta no regresa dentro de 3x del atributo `pollInterval`, el contenedor emitirá un evento `NonConsponsiveConsumerEvent`. Por defecto, esta verificación se realiza cada 30 segundos en cada contenedor. Puede modificar este comportamiento estableciendo las propiedades `monitorInterval` y `noPollThreshold` en `ContainerProperties` al configurar el contenedor de escucha. Recibir tal evento le permite detener el contenedor y despertar al consumidor para su terminación.

## Consumo de eventos

Puede capturar estos eventos implementando un `ApplicationListener`, ya sea un escucha genérico o un escucha que se reduce para recibir solo este evento en particular. También puede usar `@EventListener` introducido en Spring Framework 4.2.

El siguiente ejemplo combina `@KafkaListener` y `@EventListener` en una sola clase. Debe tener en cuenta que el escucha de la aplicación obtiene eventos para todos los contenedores, por lo que es posible que desee comprobar la ID del escucha si desea realizar una acción específica en función de qué contenedor está inactivo. También puede usar la condición `@EventListener` para este propósito.

Para obtener información sobre las propiedades del evento, vea [Eventos](#)

Este evento generalmente se publica en el hilo del consumidor, por lo que es seguro interactuar con el objeto del consumidor.

El siguiente ejemplo usa tanto `@KafkaListener` como `@EventListener`:



```
1      public class Listener {
2
3          @KafkaListener(id = "qux", topics = "annotated")
4          public void listen4(@Payload String foo, Acknowledgment ack) {
5              ...
6          }
7
8          @EventListener(condition = "event.listenerId.startsWith('qux-')")
9          public void eventHandler(ListenerContainerIdleEvent event) {
10              ...
11          }
12
13      }
```

El oyente de eventos examina los eventos para todos los contenedores. Entonces, en el ejemplo anterior, redujimos el evento recibido en función de la ID del oyente.

Dado que el contenedor creado para `@KafkaListener` admite concurrencia, el nombre real del contenedor es `id-n`, donde `n` es un valor único para cada instancia para admitir concurrencia.

Es por eso que usamos `beginWith` en la condición.

Si desea utilizar el evento inactivo para detener el contenedor `Listener`, no debe llamar a `container.stop()` en el hilo que llamó al oyente.

Hacerlo puede causar demoras y mensajes de registro innecesarios. En su lugar, debe entregar el evento a otro subproceso que pueda bloquear el contenedor. Además, si la instancia del contenedor es un contenedor secundario, no debe detenerse.

Debe detener el contenedor concurrente.

## Posiciones actuales cuando está inactivo

Tenga en cuenta que al implementar `ConsumerSeekAware` en la escucha, puede obtener la ubicación actual cuando se detecta inactivo. Consulte `Buscar onIdleContainer()` en un desplazamiento específico.

## Desplazamiento inicial de tema / partición

Varios métodos pueden establecer el desplazamiento inicial para la partición.

Cuando asigna manualmente una partición, puede establecer el desplazamiento inicial (si es necesario) en el parámetro `TopicPartitionInitialOffset` configurado (consulte Contenedor de escucha de mensajes). También puede buscar un desplazamiento específico en cualquier momento.

Cuando usa un proxy para asignar la administración de grupos particionados:



- Para el nuevo `group.id` El desplazamiento inicial está determinado por el atributo de usuario `auto.offset.reset` (más temprano o más reciente).
- Para una ID de grupo existente, el desplazamiento inicial es el desplazamiento actual de la ID de grupo. Sin embargo, puede encontrar un desplazamiento específico durante la inicialización (o en cualquier momento posterior).

## Buscando una compensación específica

Para buscar, su oyente debe implementar `ConsumerSeekAware`, que tiene los siguientes métodos:

```
1         void registerSeekCallback(ConsumerSeekCallback callback);
2
3     void onPartitionsAssigned(Map<TopicPartition, Long> assignments, ConsumerSeekCall
4
5     void onIdleContainer(Map<TopicPartition, Long> assignments, ConsumerSeekCallback
```

Se llama al primer método cuando se inicia el contenedor. Debe usar esta devolución de llamada cuando la busque en cualquier momento después de la inicialización. Debe guardar una referencia a la devolución de llamada. Si usa el mismo agente de escucha en varios contenedores (o `ConcurrentMessageListenerContainer`), debe almacenar la devolución de llamada en `ThreadLocal` o en alguna otra estructura escrita por el subproceso de escucha.

Cuando se usa la gestión de grupo, se llama al segundo método cuando se asignan los cambios. Por ejemplo, puede usar este método para establecer el desplazamiento inicial de una partición llamando a una devolución de llamada. Debe usar el parámetro de devolución de llamada en lugar del parámetro pasado a `registerSeekCallback`. Este método nunca se llama si usted asigna particiones explícitamente. En este caso, use `TopicPartitionInitialOffset`.

La devolución de llamada tiene los siguientes métodos:

```
1         void seek(String topic, int partition, long offset);
2
3         void seekToBeginning(String topic, int partition);
4
5         void seekToEnd(String topic, int partition);
```

También puede realizar una operación de búsqueda desde `onIdleContainer()` cuando se detecta un contenedor libre. Para obtener información sobre cómo habilitar la detección de contenedores inactivos, consulte [Detección de consumidores inactivos y que no responden](#).

Para buscar arbitrariamente en tiempo de ejecución, use la referencia de devolución de llamada en `registerSeekCallback` para obtener el hilo apropiado.

## Fábrica de contenedores

Como se discutió en `@KafkaListener Annotation`, `ConcurrentKafkaListenerContainerFactory` se usa para crear contenedores para métodos anotados.

A partir de la versión 2.2, puede usar la misma fábrica para crear cualquier `ConcurrentMessageListenerContainer`. Esto puede ser útil si está creando múltiples contenedores con propiedades similares, o si desea usar alguna fábrica configurada externamente, como la fábrica proporcionada por Spring Boot autoconfiguration. Una vez que se crea el contenedor, sus propiedades se pueden modificar aún más, muchas de las cuales se establecen mediante `container.getContainerProperties()`. El siguiente ejemplo configura `ConcurrentMessageListenerContainer`:

```

1      @Bean
2      public ConcurrentMessageListenerContainer<String, String>(
3          ConcurrentKafkaListenerContainerFactory<String, String> factory) {
4
5          ConcurrentMessageListenerContainer<String, String> container =
6              factory.createContainer("topic1", "topic2");
7              container.setMessageListener(m -> { ... } );
8              return container;
9          }

```

Los contenedores creados de esta manera no se agregan al registro de punto final. Deben crearse como definiciones `@Bean` para que se registren en el contexto de la aplicación.

## Hilo de seguridad

Cuando se utiliza un contenedor de escucha de mensajes concurrentes, se llama a una sola instancia de escucha en todos los subprocesos del consumidor. Por lo tanto, el oyente debe ser seguro para subprocesos y es mejor usar un oyente sin estado. Si no puede hacer que un hilo de escucha sea seguro o agregar sincronización que pueda reducir significativamente los beneficios de agregar concurrencia, puede usar una de varias técnicas:

- Utilice `n` contenedores con una concurrencia = 1 y un bean de `MessageListener` de alcance prototipo para que cada contenedor tenga su propia instancia (esto no es posible con `@KafkaListener`).
- Deje el estado en `ThreadLocal` <? > En el ejemplo.

Intimidación

- Deje que el oyente singleton delegue en el bean declarado en SimpleThreadScope (o un ámbito similar).

Para facilitar la limpieza del estado del hilo (para el segundo y el tercer elemento de la lista anterior), comenzando con la versión 2.2, el contenedor de escucha emite un `ConsumerStoppedEvent` en cada hilo que sale. Puede usar estos eventos para eliminar `ThreadLocal` del ámbito utilizando los métodos `ApplicationListener` o `@EventListener`. > Instancia o eliminar () bean de ámbito de hilo. Tenga en cuenta que `SimpleThreadScope` no destruye los beans con una interfaz destruida (como `DisposableBean`), por lo que debe destruir la instancia () usted mismo.

Por defecto, el contexto del evento de la aplicación llama al oyente del evento en el hilo de llamada. Si cambia el programa de multidifusión para usar un ejecutor asíncrono, la limpieza de subprocessos no es válida.

**30 fotos que tendrás que mirar 2 veces para entenderlas.**

NinjaJournalist

## Recomendación inteligente

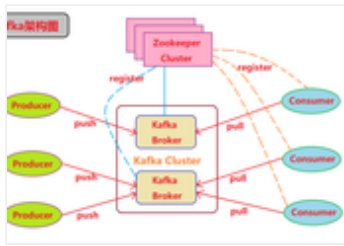
### Apache Kafka series (3) uso de API Java

Resumen: API de cliente Java Apache Kafka 1. Conceptos básicos Kafka integra la herramienta de cliente Productor / Consumidor para conectarse al intermediario, pero en términos de procesamiento de mensajes, estos dos se utilizan principalmente para ...

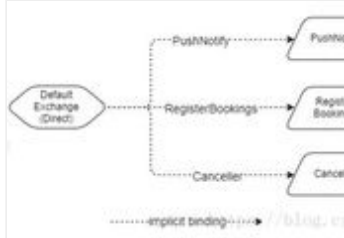
### Primavera para el uso de Apache Kafka @KafkaListener y precauciones

El documento oficial: <https://docs.spring.io/spring-kafka/reference/html/@KafkaListener> La anotación `@KafkaListener` se utiliza para designar un método de bean como escucha para un liste ...

### (3) Transmisión de datos-cola de mensajes kafka

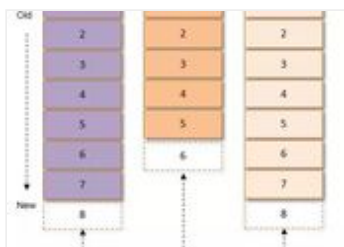


Directorio de artículos Descripción general de Kafka 1  
Introducción arquitectura de kafka 1. Diagrama de arquitectura  
2. Componentes y sus funciones Tema del agente Tema y  
partición del agente Distribución de la partición 3. Funciones y ...



## RabbitMQ VS Apache Kafka (5): topología d e enrutamiento RabbitMQ y modo de mensa je

En este capítulo, discutimos el modo de mensaje y la topología de enrutamiento de RabbitMQ, que involucra principalmente los siguientes puntos de conocimiento: Tipo de conmutador y relación de enlace cola de mensajes.



## Evolución del formato de mensaje Apache Kafka (0.7.x ~ 0.10.x)

Para un middleware de mensaje maduro, el formato del mensaje no solo está relacionado con la extensión de la dimensión funcional, sino también con la optimización de la dimensión de rendimiento. Con el rápido desarrollo ...

## Más recomendación

### Notas "Combate Apache Kafka" - 7.6.4 Ver metadatos del m ensaje

Información de metadatos Incluye desplazamiento de mensajes, marca de tiempo de creación, tipo de compresión, número de bytes y más. Primero, cree un tema de prueba y luego produzca varios mensajes: Mire el archivo de registro ...

## Spring-Kafka (9) - Configuración de filtros de mensajes



Filtro de mensajes El filtro de mensajes se puede interceptar antes de que el mensaie llegue al contenedor de escucha. El filtro filtra los datos requeridos de acu€ Intimididad

lógica comercial del sistema y luego pro ...

## Spring-boot e implemente el transmisor de mensajes kafka

1, la configuración kafakaproducer y consumidor. 2, la devolución de llamada enviando un mensaje de transmisión de manera exitosa o fallida. ...

## spring-kafka establece el tamaño del mensaje enviado

entorno spring boot2 spring cloud spring-kafka kakfa 2.2.0 Escenas Programa callspring-kakfaBuilt-inkafkaTemplateEl mensaje se envía, pero la entidad del mensaje es demasiado grande y excede el con ...

## Kafka enviar mensaje-transacción-Spring integración

- Inicio Haga clic aquí para ver esta serie de videos de apoyo Sin más preámbulos, vaya directamente al código.
- Ver más: extracto de Kafka - Declaración: Indique la fuente cuando repita ...

### El coste de los implantes dentales en Madrid podría sorprenderte

Implantes Dentales | Enlaces Publicitarios

### Artículos Relacionados

- Episodio 1 Primavera para Apache Kafka Comenzando
- Episode 2 Spring for Apache Kafka Configurando temas y enviando mensajes
- Kafka se aman matando el episodio 3-Operación Python kafka
- Primavera para el combate Apache Kafka
- Ali P8 análisis de Spring Boot y Apache Kafka combinados para lograr el manejo de errores, conversión de mensajes y soporte de transacciones?
- Spring para la documentación de Apache Kafka 2.2.6
- java.lang.NoClassDefFoundError: org / apache / kafka / common / message / KafkaKZ4BlockOutputStream



Intimidad



- [Apache Kafka-Message Queuing \(última versión\)](#)
- [nube de primavera: mensaje middleware kafka](#)
- [RabbitMQ VS Apache Kafka \(seis\): topología de enrutamiento Kafka y modo de mensaje](#)

### 35 inventos innecesarios que te sacarán una sonrisa

NinjaJournalist

Enlaces patrocinados por Taboola

#### entradas populares

- [\[2\] señala que el nuevo autodesarrollado mi primera aplicación de Android](#)
- ["El programador pragmático: desde pequeños trabajos hasta las notas de estudio de los expertos \(a\)](#)
- [Primeras imágenes de eco antes de subir](#)
- [Atributo peatonal "Aprendizaje débilmente supervisado de características de nivel medio para reconocimiento de atributos peatonales y Loca"](#)
- [1- máquina de vector de soporte SVM linealmente separable con el espacio máximo maximizado](#)
- [Diferencias y opciones entre MyISAM e InnoDB, resumen detallado, comparación de rendimiento](#)
- [Instalar el servidor mosquitto mqtt en Raspberry Pi Raspberry Pi](#)
- [2. Sistema de archivos](#)
- [Dividido](#)
- [Comprender los patrones de diseño con el código OC \(1\) Modo creado](#)



## ¿La recuerdas? No podrás creer quien es la actual pareja de su hija

Game Of Glam

## Fueron llamados los gemelos más bellos del mundo, espera hasta qu...

My Daily Magazine

Enlaces patrocinados por Taboola

### Publicaciones recomendadas

- o Inicio del clúster Kafka y secuencia de comandos de detención
- o Comunicación y valor del componente principal del niño angular
- o Gramática básica de Golang: uso detallado de variables
- o JSON
- o Cómo cancelar el menú de herramientas que aparece después de que XenDesktop5 inicia sesión.
- o Parte lógica del algoritmo PhxPaxos (1) -Introducción
- o 1.1 Editar elevación del piso
- o Tres métodos para configurar variables de entorno de Linux: / etc / profile, ~ / .bashrc, shell
- o Aprendizaje profundo y combate TensorFlow (ocho) bases de redes neuronales convolucionales
- o Qt dibuja una curva suave

### Etiquetas Relacionadas

Intimidad



Primavera para Apache Kafka

Instrucciones de actualización

Cambio de actualización

Spring Boot + Kafka

Configurar temas y enviar mensajes usando Spring para Apache Kafka

Programa de vida

Java

parte trasera

Arquitectura

Lenguaje de programación



Copyright 2018-2020 - Todos los derechos reservados -  
[www.programmersought.com](http://www.programmersought.com)

