(/)

Manejo de errores con Spring AMQP

Última modificación: 28 de diciembre de 2019

por baeldung (https://www.baeldung.com/author/baeldung/)
(https://www.baeldung.com/author/baeldung/)

Primavera (https://www.baeldung.com/category/spring/) +

Mensajería (https://www.baeldung.com/tag/messaging/)

1. Introducción

La mensajería asincrónica es un tipo de comunicación distribuida débilmente acoplada que se está volviendo cada vez más popular para implementar arquitecturas basadas en eventos (https://www.baeldung.com/cqrs-event-sourced-architecture-resources) . Afortunadamente, Spring Framework (https://www.baeldung.com/spring-intro) proporciona el proyecto Spring AMQP (https://www.baeldung.com/spring-amqp) que nos permite construir soluciones de mensajería basadas en AMQP.

Por otro lado, **lidiar con errores en tales entornos puede ser una tarea no trivial** . Entonces, en este tutorial, cubriremos diferentes estrategias para manejar errores.

2. Configuración del entorno

Para este tutorial, **utilizaremos RabbitMQ (https://www.baeldung.com/rabbitmq) que implementa el estándar AMQP.** Además, Spring AMQP proporciona el módulo *spring-rabbit* que facilita la integración.

Ejecutemos RabbitMQ como un servidor independiente. Lo ejecutaremos en un contenedor Docker (https://www.baeldung.com/dockerizing-spring-boot-application) ejecutando el siguiente comando:

```
1 docker run -d -p 5672:5672 -p 15672:15672 --name my-rabbit rabbitmq:3-management
```

Para una configuración detallada y la configuración de dependencias del proyecto, consulte nuestro artículo Spring AMQP (https://www.baeldung.com/spring-amqp) .

3. Escenario de falla

Por lo general, hay más tipos de errores que pueden ocurrir en sistemas basados en mensajería en comparación con un monolito o aplicaciones de paquete único debido a su naturaleza distribuida.

Podemos señalar algunos de los tipos de excepciones:

- *Red* o *E / S relacionadas* : fallas generales de las conexiones de red y operaciones de E / S
- *Relacionados con el protocolo* o la *infraestructura* : errores que generalmente representan una configuración incorrecta de la infraestructura de mensajería
- *Relacionado con el* agente: fallas que advierten sobre la configuración incorrecta entre clientes y un agente de AMQP. Por ejemplo, alcanzar límites o umbrales definidos, autenticación o configuración de políticas no válidas
- *Aplicaciones* y *mensajes relacionados* : excepciones que generalmente indican una violación de algunas reglas comerciales o de aplicación

Ciertamente, esta lista de fallas no es exhaustiva pero contiene el tipo más común de errores.

Debemos tener en cuenta que Spring AMQP maneja los problemas relacionados con la conexión y de bajo nivel de forma inmediata, por ejemplo, aplicando políticas de reintento o de solicitud . Además, la mayoría de las fallas y fallas se convierten en una *excepción Amqp* o una de sus subclases.

En las siguientes secciones, nos centraremos principalmente en los errores de alto nivel específicos de la aplicación y luego cubriremos las estrategias globales de manejo de errores.

4. Configuración del proyecto

Ahora, definamos una configuración simple de cola e intercambio para comenzar:

```
public static final String QUEUE_MESSAGES = "baeldung-messages-queue";
public static final String EXCHANGE_MESSAGES = "baeldung-messages-exchange";

@Bean
Queue messagesQueue() {
    return QueueBuilder.durable(QUEUE_MESSAGES)
      .build();
}

@Bean
DirectExchange messagesExchange() {
    return new DirectExchange(EXCHANGE_MESSAGES);
}

@Bean
Binding bindingMessages() {
    return BindingBuilder.bind(messagesQueue()).to(messagesExchange()).with(QUEUE_MESSAGES);
}
```

A continuación, creemos un productor simple:

```
public void sendMessage() {
    rabbitTemplate
      .convertAndSend(SimpleDLQAmqpConfiguration.EXCHANGE_MESSAGES,
        SimpleDLQAmqpConfiguration.QUEUE_MESSAGES, "Some message id:" + messageNumber++);
}
```

Y finalmente, un consumidor que arroja una excepción:

```
@RabbitListener(queues = SimpleDLQAmqpConfiguration.QUEUE_MESSAGES)
public void receiveMessage(Message message) throws BusinessException {
    throw new BusinessException();
}
```

Por defecto, todos los mensajes fallidos se volverán a solicitar inmediatamente en la parte superior de la cola de destino una y otra vez.

Ejecutemos nuestra aplicación de muestra ejecutando el siguiente comando Maven:

```
mvn spring-boot:run -Dstart-class=com.baeldung.springamqp.errorhandling.ErrorHandlingApp
```

Ahora deberíamos ver la salida resultante similar:

```
WARN 22260 --- [ntContainer#0-1] s.a.r.l.ConditionalRejectingErrorHandler :
  Execution of Rabbit message listener failed.
Caused by: com.baeldung.springamqp.errorhandling.errorhandler.BusinessException: null
```

En consecuencia, de forma predeterminada, veremos un número infinito de tales mensajes en la salida.

Para cambiar este comportamiento tenemos dos opciones:

- Establezca la opción *default-requeue -jected* en *false* en el lado del oyente - *spring.rabbitmq.listener.simple.default-requeue -jected = false*
- Lanza una *excepción AmqpRejectAndDontRequeueException; esta* podría ser útil para mensajes que no tendrán sentido en el futuro, de modo que puedan descartarse.

Ahora, descubramos cómo procesar mensajes fallidos de una manera más inteligente.

5. Cola de letra muerta

Una Cola de letras muertas (DLQ) es una cola que contiene mensajes no entregados o fallidos . Un DLQ nos permite manejar mensajes defectuosos o incorrectos, monitorear patrones de falla y recuperar excepciones de un sistema.

Más importante aún, esto ayuda a evitar bucles infinitos en las colas que constantemente procesan mensajes erróneos y degradan el rendimiento del sistema.

En total, hay dos conceptos principales: Dead Letter Exchange (DLX) y una Dead Letter Queue (DLQ). De hecho, **DLX es un intercambio normal que podemos definir como uno de los tipos comunes** : *directo* , *tema* o *fanout* .

Es muy importante entender que **un productor no sabe nada sobre las colas. Solo es consciente de los intercambios y todos los mensajes producidos se enrutan de acuerdo con la configuración del intercambio y la clave de enrutamiento de mensajes** .

Ahora veamos cómo manejar las excepciones aplicando el enfoque de la Cola de letra muerta.

5.1. Configuracion basica

Para configurar un DLQ necesitamos especificar argumentos adicionales al definir nuestra cola:

```
@Bean
Queue messagesQueue() {
    return QueueBuilder.durable(QUEUE_MESSAGES)
      .withArgument("x-dead-letter-exchange", "")
      .withArgument("x-dead-letter-routing-key", QUEUE_MESSAGES_DLQ)
      .build();
}

@Bean
Queue deadLetterQueue() {
    return QueueBuilder.durable(QUEUE_MESSAGES_DLQ).build();
}
```

En el ejemplo anterior, hemos usado dos argumentos adicionales: *x-dead-letter-exchange* y *x-dead-letter-routing-key* . **El valor de cadena vacía para la opción *x-dead-letter-exchange* le dice al corredor que use el intercambio predeterminado** .

El segundo argumento es tan importante como establecer claves de enrutamiento para mensajes simples. Esta opción cambia la clave de enrutamiento inicial del mensaje para su posterior enrutamiento por DLX.

5.2. Enrutamiento de mensajes fallidos

Entonces, cuando un mensaje no se entrega, se enruta al Intercambio de letras muertas. Pero como ya hemos señalado, **DLX es un intercambio normal. Por lo tanto, si la clave de enrutamiento del mensaje fallido no coincide con el intercambio, no se entregará a la DLQ.**

```
Exchange: (AMQP default)
Routing Key: baeldung-messages-queue.dlq
```

Entonces, si omitimos el argumento *x-dead-letter-routing-key* en nuestro ejemplo, el mensaje fallido quedará atascado en un ciclo de reintento infinito.

Además, la metainformación original del mensaje está disponible en el encabezado *x-death* :

```
x-death:
  count: 1
  exchange: baeldung-messages-exchange
  queue: baeldung-messages-queue
  reason: rejected
  routing-keys: baeldung-messages-queue
  time: 1571232954
```

La **información anterior está disponible en la consola de administración RabbitMQ que** generalmente se ejecuta localmente en el puerto 15672.

Además de esta configuración, si estamos utilizando Spring Cloud Stream (https://www.baeldung.com/spring-cloud-stream) , incluso podemos simplificar el proceso de configuración aprovechando las propiedades de configuración *republishToDlq* y *autoBindDlq* .

5.3. Intercambio de letras muertas

En la sección anterior, hemos visto que la clave de enrutamiento cambia cuando un mensaje se enruta al intercambio de letras muertas. Pero este comportamiento no siempre es deseable. Podemos cambiarlo configurando DLX por nosotros mismos y definiéndolo usando el tipo *fanout* :

```
public static final String DLX_EXCHANGE_MESSAGES = QUEUE_MESSAGES + ".dlx";

@Bean
Queue messagesQueue() {
    return QueueBuilder.durable(QUEUE_MESSAGES)
      .withArgument("x-dead-letter-exchange", DLX_EXCHANGE_MESSAGES)
      .build();
}

@Bean
FanoutExchange deadLetterExchange() {
    return new FanoutExchange(DLX_EXCHANGE_MESSAGES);
}

@Bean
Queue deadLetterQueue() {
    return QueueBuilder.durable(QUEUE_MESSAGES_DLQ).build();
}

@Bean
Binding deadLetterBinding() {
    return BindingBuilder.bind(deadLetterQueue()).to(deadLetterExchange());
}
```

Esta vez hemos definido un intercambio personalizado del tipo de *despliegue* , por lo que se enviarán mensajes a todas las colas delimitadas . Además, hemos establecido el valor del argumento *x-dead-letter-exchange* para el nombre de nuestro DLX. Al mismo tiempo, hemos eliminado el argumento *x-dead-letter-routing-key* .

Ahora, si ejecutamos nuestro ejemplo, el mensaje fallido debería enviarse al DLQ, pero sin cambiar la clave de enrutamiento inicial:

```
Exchange: baeldung-messages-queue.dlx
Routing Key: baeldung-messages-queue
```

5.4. Procesamiento de mensajes de cola de letra muerta

Por supuesto, la razón por la que los trasladamos a Dead Letter Queue es para que puedan ser reprocesados en otro momento.

Definamos un oyente para Dead Letter Queue:

```
@RabbitListener(queues = QUEUE_MESSAGES_DLQ)
public void processFailedMessages(Message message) {
    log.info("Received failed message: {}", message.toString());
}
```

Si ejecutamos nuestro código de ejemplo ahora, deberíamos ver la salida del registro:

```
WARN 11752 --- [ntContainer#0-1] s.a.r.l.ConditionalRejectingErrorHandler :
  Execution of Rabbit message listener failed.
INFO 11752 --- [ntContainer#1-1] c.b.s.e.consumer.SimpleDLQAmqpContainer  :
  Received failed message:
```

Tenemos un mensaje fallido, pero ¿qué debemos hacer a continuación? La respuesta depende de los requisitos específicos del sistema, el tipo de excepción o el tipo de mensaje.

Por ejemplo, podemos enviar el mensaje al destino original:

```
@RabbitListener(queues = QUEUE_MESSAGES_DLQ)
public void processFailedMessagesRequeue(Message failedMessage) {
    log.info("Received failed message, requeueing: {}", failedMessage.toString());
    rabbitTemplate.send(EXCHANGE_MESSAGES,
      failedMessage.getMessageProperties().getReceivedRoutingKey(), failedMessage);
}
```

Pero dicha lógica de excepción no es diferente de la política de reintento predeterminada:

```
INFO 23476 --- [ntContainer#0-1] c.b.s.e.c.RoutingDLQAmqpContainer        :
  Received message:
WARN 23476 --- [ntContainer#0-1] s.a.r.l.ConditionalRejectingErrorHandler :
  Execution of Rabbit message listener failed.
INFO 23476 --- [ntContainer#1-1] c.b.s.e.c.RoutingDLQAmqpContainer        :
  Received failed message, requeueing:
```

Una estrategia común puede necesitar volver a intentar procesar un mensaje por *n* veces y luego rechazarlo. Implementemos esta estrategia aprovechando los encabezados de mensajes:

```
public void processFailedMessagesRetryHeaders(Message failedMessage) {
    Integer retriesCnt = (Integer) failedMessage.getMessageProperties()
      .getHeaders().get(HEADER_X_RETRIES_COUNT);
    if (retriesCnt == null) retriesCnt = 1;
    if (retriesCnt > MAX_RETRIES_COUNT) {
        log.info("Discarding message");
        return;
    }
    log.info("Retrying message for the {} time", retriesCnt);
    failedMessage.getMessageProperties()
      .getHeaders().put(HEADER_X_RETRIES_COUNT, ++retriesCnt);
    rabbitTemplate.send(EXCHANGE_MESSAGES,
      failedMessage.getMessageProperties().getReceivedRoutingKey(), failedMessage);
}
```

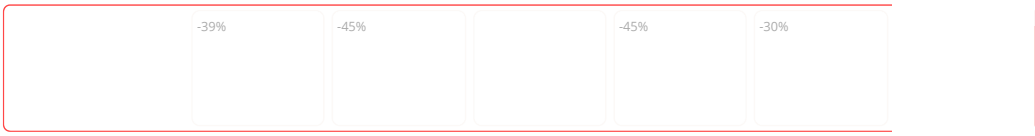
Al principio, estamos obteniendo el valor del encabezado *x-retries-count* , luego comparamos este valor con el valor máximo permitido. Posteriormente, si el contador alcanza el número límite de intentos, el mensaje será descartado:

```
WARN 1224 --- [ntContainer#0-1] s.a.r.l.ConditionalRejectingErrorHandler :
  Execution of Rabbit message listener failed.
INFO 1224 --- [ntContainer#1-1] c.b.s.e.consumer.DLQCustomAmqpContainer  :
  Retrying message for the 1 time
WARN 1224 --- [ntContainer#0-1] s.a.r.l.ConditionalRejectingErrorHandler :
  Execution of Rabbit message listener failed.
INFO 1224 --- [ntContainer#1-1] c.b.s.e.consumer.DLQCustomAmqpContainer  :
  Retrying message for the 2 time
WARN 1224 --- [ntContainer#0-1] s.a.r.l.ConditionalRejectingErrorHandler :
  Execution of Rabbit message listener failed.
INFO 1224 --- [ntContainer#1-1] c.b.s.e.consumer.DLQCustomAmqpContainer  :
  Discarding message
```

Debemos agregar que también podemos hacer uso del encabezado *x-message-ttl* para establecer un tiempo después de que el mensaje deba descartarse. Esto podría ser útil para evitar que las colas crezcan infinitamente.

5.5. Cola de estacionamiento

Por otro lado, considere una situación en la que no podemos simplemente descartar un mensaje, podría ser una transacción en el dominio bancario, por ejemplo. Alternativamente, a veces un mensaje puede requerir procesamiento manual o simplemente necesitamos grabar mensajes que fallaron más de *n* veces.



Para situaciones como esta, existe **un concepto de una cola de estacionamiento** . Podemos **reenviar todos los mensajes del DLQ, que fallaron más de la cantidad permitida de veces, a la Cola del estacionamiento para su posterior procesamiento** .

Implementemos ahora esta idea:

```
public static final String QUEUE_PARKING_LOT = QUEUE_MESSAGES + ".parking-lot";
public static final String EXCHANGE_PARKING_LOT = QUEUE_MESSAGES + "exchange.parking-lot";

@Bean
FanoutExchange parkingLotExchange() {
    return new FanoutExchange(EXCHANGE_PARKING_LOT);
}

@Bean
Queue parkingLotQueue() {
    return QueueBuilder.durable(QUEUE_PARKING_LOT).build();
}

@Bean
Binding parkingLotBinding() {
    return BindingBuilder.bind(parkingLotQueue()).to(parkingLotExchange());
}
```

En segundo lugar, refactoricemos la lógica del oyente para enviar un mensaje a la cola del estacionamiento:

```
@RabbitListener(queues = QUEUE_MESSAGES_DLQ)
public void processFailedMessagesRetryWithParkingLot(Message failedMessage) {
    Integer retriesCnt = (Integer) failedMessage.getMessageProperties()
      .getHeaders().get(HEADER_X_RETRIES_COUNT);
    if (retriesCnt == null) retriesCnt = 1;
    if (retriesCnt > MAX_RETRIES_COUNT) {
        log.info("Sending message to the parking lot queue");
        rabbitTemplate.send(EXCHANGE_PARKING_LOT,
          failedMessage.getMessageProperties().getReceivedRoutingKey(), failedMessage);
        return;
    }
    log.info("Retrying message for the {} time", retriesCnt);
    failedMessage.getMessageProperties()
      .getHeaders().put(HEADER_X_RETRIES_COUNT, ++retriesCnt);
    rabbitTemplate.send(EXCHANGE_MESSAGES,
      failedMessage.getMessageProperties().getReceivedRoutingKey(), failedMessage);
}
```

Finalmente, también debemos procesar los mensajes que llegan a la cola del estacionamiento:

```
@RabbitListener(queues = QUEUE_PARKING_LOT)
public void processParkingLotQueue(Message failedMessage) {
    log.info("Received message in parking lot queue");
    // Save to DB or send a notification.
}
```

Ahora podemos guardar el mensaje fallido en la base de datos o tal vez enviar una notificación por correo electrónico.

Probemos esta lógica ejecutando nuestra aplicación:

```
WARN 14768 --- [ntContainer#0-1] s.a.r.l.ConditionalRejectingErrorHandler :
  Execution of Rabbit message listener failed.
INFO 14768 --- [ntContainer#1-1] c.b.s.e.c.ParkingLotDLQAmqpContainer     :
  Retrying message for the 1 time
WARN 14768 --- [ntContainer#0-1] s.a.r.l.ConditionalRejectingErrorHandler :
  Execution of Rabbit message listener failed.
INFO 14768 --- [ntContainer#1-1] c.b.s.e.c.ParkingLotDLQAmqpContainer     :
  Retrying message for the 2 time
WARN 14768 --- [ntContainer#0-1] s.a.r.l.ConditionalRejectingErrorHandler :
  Execution of Rabbit message listener failed.
INFO 14768 --- [ntContainer#1-1] c.b.s.e.c.ParkingLotDLQAmqpContainer     :
  Sending message to the parking lot queue
INFO 14768 --- [ntContainer#2-1] c.b.s.e.c.ParkingLotDLQAmqpContainer     :
  Received message in parking lot queue
```

Como podemos ver en la salida, después de varios intentos fallidos, el mensaje se envió a la cola del estacionamiento.

6. Manejo de errores personalizado

En la sección anterior, hemos visto cómo manejar fallas con colas e intercambios dedicados. **Sin embargo, a veces es posible que necesitemos detectar todos los errores, por ejemplo para iniciar sesión o persistirlos en la base de datos.**

6.1. Global *ErrorHandler*

Hasta ahora, hemos usado el *SimpleRabbitListenerContainerFactory* predeterminado y esta fábrica por defecto usa *ConditionalRejectingErrorHandler* . Este controlador detecta diferentes excepciones y las transforma en una de las excepciones dentro de la jerarquía *AmqpException* .

Es importante mencionar que si necesitamos manejar errores de conexión, entonces necesitamos implementar la interfaz *ApplicationListener* .

En pocas palabras, ***ConditionalRejectingErrorHandler* decide si rechazar un mensaje específico o no.** Cuando se rechaza el mensaje que causó una excepción, no será solicitado.

Definamos un *ErrorHandler* personalizado que simplemente requiera solo *BusinessException* s:

```
public class CustomErrorHandler implements ErrorHandler {
    @Override
    public void handleError(Throwable t) {
        if (!(t.getCause() instanceof BusinessException)) {
            throw new AmqpRejectAndDontRequeueException("Error Handler converted exception to fatal", t);
        }
    }
}
```

Además, como estamos lanzando la excepción dentro de nuestro método de escucha, está envuelta en una *ListenerExecutionFailedException* . Por lo tanto, debemos llamar al método *getCause* para obtener una excepción de origen.

6.2. *FatalExceptionStrategy*

Debajo del capó, este controlador utiliza la *estrategia FatalExceptionStrategy* para verificar si una excepción debe considerarse fatal. Si es así, el mensaje fallido será rechazado.

Por defecto, estas excepciones son fatales:

- *MessageConversionException*
- *MessageConversionException*
- *MethodArgumentNotValidException*

- *MethodArgumentTypeMismatchException*
- *NoSuchMethodException*
- *ClassCastException*

En lugar de implementar la interfaz *ErrorHandler*, podemos proporcionar nuestra *estrategia FatalExceptionStrategy*:

```
public class CustomFatalExceptionStrategy
      extends ConditionalRejectingErrorHandler.DefaultExceptionStrategy {
    @Override
    public boolean isFatal(Throwable t) {
        return !(t.getCause() instanceof BusinessException);
    }
}
```

Finalmente, necesitamos pasar nuestra estrategia personalizada al constructor *ConditionalRejectingErrorHandler*:

```
@Bean
public SimpleRabbitListenerContainerFactory rabbitListenerContainerFactory(
  ConnectionFactory connectionFactory,
  SimpleRabbitListenerContainerFactoryConfigurer configurer) {
      SimpleRabbitListenerContainerFactory factory =
        new SimpleRabbitListenerContainerFactory();
      configurer.configure(factory, connectionFactory);
      factory.setErrorHandler(errorHandler());
      return factory;
}

@Bean
public ErrorHandler errorHandler() {
    return new ConditionalRejectingErrorHandler(customExceptionStrategy());
}

@Bean
FatalExceptionStrategy customExceptionStrategy() {
    return new CustomFatalExceptionStrategy();
}
```

7. Conclusión

En este tutorial, hemos discutido diferentes formas de manejar errores al usar Spring AMQP, y RabbitMQ en particular.

Cada sistema necesita una estrategia específica de manejo de errores. Hemos cubierto las formas más comunes de manejo de errores en arquitecturas controladas por eventos. Además, hemos visto que podemos combinar múltiples estrategias para construir una solución más completa y robusta.

Como siempre, el código fuente completo del artículo está disponible en GitHub (https://github.com/eugenp/tutorials/tree/master/spring-amqp) .

Deja una respuesta

CATEGORÍAS

PRIMAVERA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
DESCANSO (HTTPS://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SEGURIDAD (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCIA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JSON/JACKSON/)
HTTP DEL LADO DEL CLIENTE (HTTPS://WWW.BAELDUNG.COM/CATEGORY/HTTP/)
KOTLIN (HTTPS://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/)

SERIE

TUTORIAL DE JAVA "VOLVER A LO BÁSICO" (/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (/JACKSON)
HTTPCLIENT 4 TUTORIAL (/HTTPCLIENT-GUIDE)
RESTO CON SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)
TUTORIAL SPRING PERSISTENCE (/PERSISTENCE-WITH-SPRING-SERIES)
SEGURIDAD CON PRIMAVERA (/SECURITY-SPRING)

ACERCA DE

SOBRE BAELDUNG (/ABOUT)
LOS CURSOS (HTTPS://COURSES.BAELDUNG.COM)
TRABAJO DE CONSULTORÍA (/CONSULTING)
META BAELDUNG (HTTP://META.BAELDUNG.COM/)
EL ARCHIVO COMPLETO (/FULL_ARCHIVE)
ESCRIBIR PARA BAELDUNG (/CONTRIBUTION-GUIDELINES)
EDITORES (/EDITORS)
NUESTROS COMPAÑEROS (/PARTNERS)
ANUNCIE EN BAELDUNG (/ADVERTISE)

TÉRMINOS DE SERVICIO (/TERMS-OF-SERVICE)
POLÍTICA DE PRIVACIDAD (/PRIVACY-POLICY)
INFORMACIÓN DE LA COMPAÑÍA (/BAELDUNG-COMPANY-INFO)
CONTACTO (/CONTACT)