DeadLetterQueue 재처리와 Retry

Dead Letter Queue와 Dead Letter Exchange

DLQ : 메시지가 큐에서 제대로 처리 되지 못할 경우 DLQ에 이동 되며, 실패한 메시지를 저장하는 용도로 사용

- NACK 처리나 거부
  - basic.reject 혹은 basic.nack 으로 메시지가 처리되지 못한 경우
- TTL (Time-To-Live) 만료
  - 메시지 TTL이 초과된 경우 DLQ로 이동
- 큐 설정 초과 (Overflow)
  - 큐에 설정된 최대 메시지 갯수를 초과하면 가장 오래된 메시지가 삭제되고 DLQ로 이동

DLX : 큐 실패시에 데드레터 익스체인지(DLX)를 설정하여 메시지가 처리되지 못한 경우 지정된 큐로 이동시킬수 있다.

예를들어, 잘못된 형식의 큐로 인해 처리가 어려운 경우 Dead Letter Queue 영역으로 이동하여 에러의 원인을 분석하고 재 처리를 시도할 수 있다.

이런 경우는 처리 중에 예외가 발생한 메시지, TTL(Time To Live)이 만료된 메시지, 큐의 길이 제한(x-max-length)을 초과하여 삭제된 메시지 등이 DLX로 전달되고, Dead Letter Queue에 있는 메시지를 통해 에러의 원인을 분석할 수 있다.

```
@Bean
public Queue orderQueue() {
    return QueueBuilder.durable(MAIN_QUEUE)
            .withArgument("x-dead-letter-exchange", DLX) // DLX
            .withArgument("x-dead-letter-routing-key", DLQ) // D
            .build();
}
```

튜토리얼 Step 7. Dead Letter Queue를 활용한 실패 메시지 재처리

개발 프로세스

1. DLQ와 DLX 빈 선언
2. SimpleRabbitListenerContainerFactory를 이용하여 AcknowledgeMode 모드를 Manual로 설정
    a. Ack/Nack 처리를 개발 시점에 직접 핸들링 하도록 설정
3. Producer 메시지 발행
4. Consumer 에서 메시지 실패건에 대해 (channel.basicReject) 3번의 재시도 후 DLQ 이동 (channel.basicNack)
5. 성공일 경우 Ack 전송 (channel.basicAck)
6. REST API로 성공과 실패건에 대한 테스트 진행

DLQ 재처리 Package Structure

```
src/main/java/net/harunote/hellomessagequeue git:[tutorial-step5
├── HelloController.java
├── HelloMessageQueueApplication.java
└── step7
    ├── OrderDLQConsumer.java
    ├── OrderConsumer.java
    ├── OrderController.java
    ├── OrderProducer.java
    ├── RabbitMQConfig.java
    └── RabbitMQManualConfig.java
```

이 예제를 통해 `SimpleRabbitListenerContainerFactory`의 **수동 Ack 모드 설정**
(`(AcknowledgeMode.MANUAL)`을 통해 메시지의 처리 결과를 RabbitMQ에 명시적으로 전달하고

`basicAck`, `basicNack`, `basicReject`를 사용하여 메시지 재시도 및 DLQ 처리를 유연하게 구현할 수 있다.

channel.basicAck : 메시지가 성공적으로 처리되었다는 것을 리턴

- 메시지가 Ack 되면 메시지를 큐에서 삭제하고 다른 소비자에게 전송하지 않음
- `channel.basicAck(tag, false); // Ack 전송`
  - deliveryTag(long) 필드는 메시지 고유식별 태그
  - multiple(boolean) 필드는 true 일 경우 이전의 모든 메시지를 한꺼번에 Ack 처리, false 일 경우 현재 태그의 메시지 하나만 Ack 처리
- `channel.basicNack(tag, false, false); // DLQ로 메시지 이동`
  - deliveryTag, multiple, requeue : basicAck와 동일하며 requeue (boolean)의 경우 true 일 경우 메시지를 큐에 다시 넣어 재처리 하도록 설정, false 의 경우 메시지를 DLQ로 이동 또는 삭제
- `channel.basicReject(deliveryTag, requeue);`
  - 파라미터 동일

다만 이 처리가 다소 복잡하고, 메서드 호출도 혼동되기 쉬우므로 이런 처리보다는 Spring AMQP에서 제공하는 RetryTemplate을 통해 좀더 명확하고 간단하게 기능을 구현할 수 있다.

단, RabbitMQ에서 메시지 처리 실패를 관리하기 위해 수동으로 `Ack`와 `Nack`을 호출하는 방식은 **RabbitMQ의 저수준 API**를 명확히 이해를 전제로 하기 때문에 굉장히 세밀한 애플리케이션에는 적용할 수 있겠지만 Production 에서는 유지보수가 어렵다.

이 경우 Spring AMQP에서 AcknowledgeMode가 AUTO로 기본 세팅되어 있기 때문에 별도의 SimpleRabbitListenerContainerFactory를 통하지 않고 자동으로 Ack/Nack 처리가 가능하다.

따라서, RetryTemplate을 통해 좀 더 로직을 쉽게 관리할 수 있도록 추상화된 방식으로 다음 튜토리얼을 참고하도록 한다.

튜토리얼 Step 8-1. RetryTemplate을 통한 간편한 재처리 설정

동일한 프로세스로 SimpleRetryPolicy를 설정한 뒤에 RetryTemplate에 담아 처리하는 방식으로 개선한다.

자동 재시도 처리

1. `RetryTemplate`:
    - Spring AMQP는 `RetryTemplate`을 통해 재시도 로직을 지원
    - 최대 3번 재시도 후에도 실패하면 Spring이 메시지를 DLQ로 이동
2. `AcknowledgeMode.AUTO`: (기본 설정)
    - 재시도 중 메시지가 성공적으로 처리되면 Spring AMQP가 자동으로 Ack를 전송
    - 모든 재시도가 실패하면 Nack를 보내고 RabbitMQ가 메시지를 DLQ로 이동
    - DLQ에서 메시지를 수정한 뒤 원래 큐로 재전송하여 정상 처리
3. build.gradle 에 retry 디펜던시 추가 :

```
implementation 'org.springframework.retry:spring-retry'
```

4. RetryConfig 클래스에서 기본 설정 세팅하여 빈으로 선언

```
@Configuration
public class RetryConfig {

    @Bean
    public RetryTemplate retryTemplate() {
        RetryTemplate retryTemplate = new RetryTemplate();
        // 재시도 정책 설정: 최대 3번 시도
        SimpleRetryPolicy retryPolicy = new SimpleRetryPolicy()
        retryPolicy.setMaxAttempts(3);
        // 백오프 정책 설정: 재시도 간격 1초
        FixedBackOffPolicy backOffPolicy = new FixedBackOffPoli
        backOffPolicy.setBackOffPeriod(1000L);
        retryTemplate.setRetryPolicy(retryPolicy);
        retryTemplate.setBackOffPolicy(backOffPolicy);

        return retryTemplate;
```

```
    }
}
```

4. `OrderConsumer` 3번의 retry 후 DLQ에 전송(routing key와 함께)
    a. RabbitMQ로 부터 메시지를 수신한 후 retryTemplate.execute() 내에서 메시지를 처리
    b. 메시지가 fail 일 경우 RuntimeException이 발생하고 catch 블록으로 이전
        i. retryTemplate의 retryCount를 확인하여 세번 이하라면 throw e를 던짐으로써 `(channel.basicReject(tag, true)`, RetryTemplate이 예외를 감지해서 BackOffPolicy에 따라 1초 후 네번까지 재실행(동일한 로직 실행) 후
        ii. getRetryCount가 3보다 크면 DLQ로 이관
    c. 재시도가 중단되면 메시지는 DLQ로 이관되거나 내부적으로 Ack/Nack 처리됨 (성공일 경우 `channel.basicAck(tag, false);` , 에러일 경우 `channel.basicNack(tag, false, false);` 와 내부적으로 동일)
    d. retryCount가 충족되면 DLQ로 이관되고 재시도가 중단

```
retryTemplate.execute(context -> {
      try {
          System.out.println("# 리시브 메시지 : " + message + " #
          // 실패 조건
          if ("fail".equalsIgnoreCase(message)) {
              throw new RuntimeException(message);
          }
          System.out.println("# 메시지 처리 성공 " + message);
      } catch (Exception e) {
          if (context.getRetryCount() >= 2) {
              rabbitTemplate.convertAndSend(RabbitMQConfig.ORDEI
                      RabbitMQConfig.DEAD_LETTER_ROUTING_KEY, me
          } else {
              throw e;
          }
      }
```

```
      return null;
    });
```

4. DLQ 에서 로직을 수정한 이후 다시 원래 큐로 재전송
4. 정상 처리

튜토리얼 Step 8-2. Application.yml 설정을 통한 RetryTemplate 속성 정의

1. application.yml 에 rabbitmq listener retry 설정 추가

```
spring:
  rabbitmq:
    host: localhost
    port: 5672
    username: guestuser
    password: guestuser
    listener:
      simple:
        retry:
          enabled: true # 재시도 활성화
          initial-interval: 1000 # 첫 재시도 대기 시간 1초
          max-attempts: 3 # 최대 재시도 횟수
          max-interval: 1000 # 시도간 최대 대기시간
        default-requeue-rejected: false # 재시도 실패 시 자동으로 DL
```

2. OrderConsumer 클래스 로직 수정

```
@Component
public class OrderConsumer {

    private int retryCount;

    @RabbitListener(queues = RabbitMQConfig.ORDER_COMPLETED_QUEU
```

```
    public void processMessage(String message) {
        System.out.println("Received message: "+ message + "cou
        if ("fail".equals(message)) {
            throw new RuntimeException("- Processing failed. Re
        }
        System.out.println("Message processed successfully: " +
        retryCount = 0;
    }
}
```

이와같이 설정의 변화를 통해 클래스간의 결합도를 최소화 하여 깔끔한 로직을 구현해줄 수 있다.