示例说明

此示例演示Virtual Hosts和权限的使用，及客户端链接集群的用法。

Virtual Hosts

每一个 RabbitMQ 服务器都能创建虚拟的消息服务器，我们称之为虚拟主机 (virtual host) ,简称为 vhost。

每一个 vhost 本质上是一个独立的小型 RabbitMQ 服务器，拥有自己独立的队列、交换器及绑定关系等，并且它拥有自己独立的权限。

vhost 就像是虚拟机与物理服务器一样，它们在各个实例间提供逻辑上的分离，为不同程序安全保密地运行数据，它既能将同一个RabbitMQ 中的众多客户区分开，又可以避免队列和交换器等命名冲突。

vhost 之间是绝对隔离的，无法将 vhostl 中的交换器与 vhost2 中的队列进行绑定，这样既保证了安全性，又可以确保可移植性。

如果在使用 RabbitMQ 达到一定规模的时候，建议用户对业务功能、场景进行归类区分，并为之分配独立的 vhost。

Virtual Hosts 的功能说明

vhost可以限制最大连接数和最大队列数，并且可以设置vhost下的用户资源权限和Topic权限，具体权限见下方说明。

- 在 `Admin -> Limits` 页面可以设置vhost的最大连接数和最大队列数，达到限制后，继续创建，将会报错。
- 用户资源权限是指RabbitMQ 用户在客户端执行AMQP操作命令时，拥有对资源的操作和使用权限。权限分为三个部分：`configure、write、read`，见下方表格说明。参考：http://www.rabbitmq.com/access-control.html#permissions

AMQP 0-9-1 Operation		configure	write	read
exchange.declare	(passive=false)	exchange		
exchange.declare	(passive=true)			
exchange.declare	(with AE)	exchange	exchange (AE)	exchange
exchange.delete		exchange		
queue.declare	(passive=false)	queue		
queue.declare	(passive=true)			
queue.declare	(with DLX)	queue	exchange (DLX)	queue
queue.delete		queue		
exchange.bind			exchange (destination)	exchange (source)
exchange.unbind			exchange (destination)	exchange (source)
queue.bind			queue	exchange
queue.unbind			queue	exchange
basic.publish			exchange	
basic.get				queue
basic.consume				queue
queue.purge				queue

举例说明：

- 比如创建队列时，会调用`queue.declare`方法，此时会使用到`configure`权限，会校验队列名是否与`configure`的表达式匹配。
- 比如队列绑定交换器时，会调用`queue.bind`方法，此时会用到`write`和`read`权限，会检验队列名是否与`write`的表达式匹配，交换器名是否与`read`的表达式匹配。

- Topic权限，参考：http://www.rabbitmq.com/access-control.html#topic-authorisation
  - Topic权限是RabbitMQ 针对STOMP和MQTT等协议实现的一种权限。由于这类协议都是基于Topic消费的，而AMQP是基于Queue消费，所以AMQP的标准资源权限不适合用在这类协议中，而Topic权限也不适用于AMQP协议。所以，我们一般不会去使用它，只用在使用了MQTT这类的协议时才可能会用到。

vhost使用示例

1. 使用管理员用户登录Web管理界面。
2. 在`Admin -> Virtual Hosts`页面添加一个名为`v1`的Virtual Hosts。
   - 此时还需要为此vhost分配用户，添加一个新用户
3. 在`Admin -> Users`页面添加一个名为`order-user`的用户，并设置为`management`角色。
4. 从`Admin`进入`order-user`的用户设置界面,在`Permissions`中，为用户分配vhost为/v1，并为每种权限设置需要匹配的目标名称的正则表达式。

字段名	值	说明
Virtual Host	/v1	指定用户的vhost，以下权限都只限于 `/v1` vhost中
Configure regexp	eq-.*	只能操作名称以eq-开头的exchange或queue；为空则不能操作任何exchange和queue
Write regexp	.*	能够发送消息到任意名称的exchange，并且能绑定到任意名称的队列和任意名称的目标交换器（指交换器绑定到交换器），为空表示没有权限
Read regexp	^test$	只能消费名为test队列上的消息，并且只能绑定到名为test的交换器

5. 执行示例代码 `VirtualHostsDemo`。

集群连接恢复

- **参考**：https://www.rabbitmq.com/api-guide.html#connection-recovery
- 通过 `factory.setAutomaticRecoveryEnabled(true);` 可以设置连接自动恢复的开关，默认已开启
- 通过 `factory.setNetworkRecoveryInterval(10000);` 可以设置间隔多长时间尝试恢复一次，默认是5秒：`com.rabbitmq.client.ConnectionFactory.DEFAULT_NETWORK_RECOVERY_INTERVAL`
- 什么时候会触发连接恢复？https://www.rabbitmq.com/api-guide.html#recovery-triggers
  - 如果启用了自动连接恢复，将由以下事件触发：
    - 连接的I/O循环中抛出IOExceiption
    - 读取Socket套接字超时
    - 检测不到服务器心跳
    - 在连接的I/O循环中引发任何其他异常
  - 如果客户端第一次连接失败，不会自动恢复连接。需要我们自己负责重试连接、记录失败的尝试、实现重试次数的限制等等。
    ```java
    ConnectionFactory factory = new ConnectionFactory();
    // 设置连接配置
    ```

    ```
    try {
        Connection conn = factory.newConnection();
    } catch (java.net.ConnectException e) {
        Thread.sleep(5000);
        // 重新连接
    }
    ```
  - 如果程序中调用了 `Connection.Close`，也不会自动恢复连接。
  - 如果是 `Channel-level` 的异常，也不会自动恢复连接，因为这些异常通常是应用程序中存在语义问题(例如试图从不存在的队列消费)。
- 在Connection和Channel上，可以设置重新连接的监听器，开始重连和重连成功时，会触发监听器。添加和移除监听，需要将Connection或Channel强制转换成Recoverable接口。

```
((Recoverable) connection).addRecoveryListener()
((Recoverable) connection).removeRecoveryListener()
```

重连测试方式

- 测试前，先按集群部署方式搭建好集群。
- 开启集群节点后，启动`Consumer`和`Producer`。
- 使用`rabbitmqctl -n [node_name] stop_app`命令关闭一个节点，例如：`rabbitmqctl -n rabbit2 stop_app`;
- 查看Consumer和Producer控制台是否有重连的信息。
- 使用`rabbitmqctl -n [node_name] start_app`可开启关闭的节点。

镜像队列测试

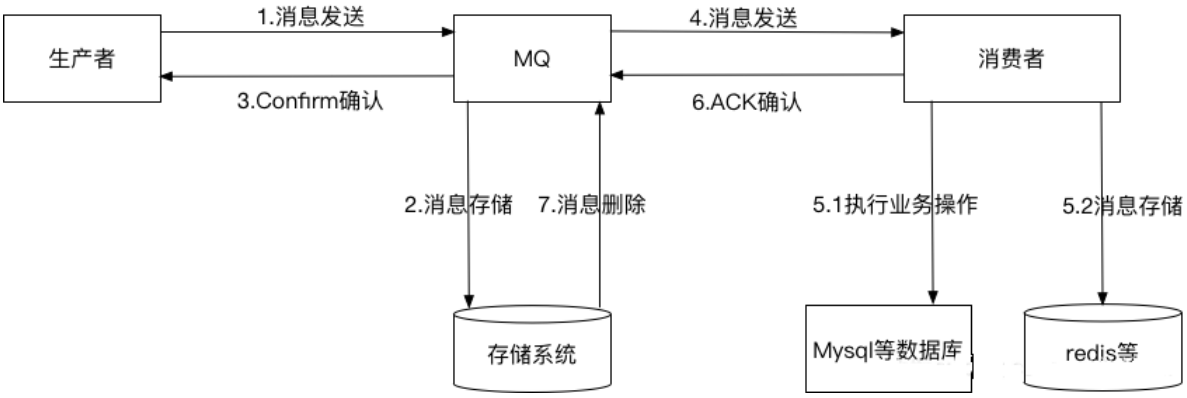
- 测试方式
  - 生产者连接10.10.1.41:5672发送消息后，停止rabbit1节点

队列持久化	消息持久化	镜像队列	结果
否	否	否	rabbit1重启后，队列和消息丢失
是	否	否	rabbit1重启后，队列存在但消息丢失；rabbit1不启动消费者连接其它节点也无法启动
是	是	否	rabbit1重启后，队列存在，消息丢失；rabbit1不启动消费者连接其它节点也无法启动
否	否	是	对列和消息都还存在，并且消费者能够正常消费
是	否	是	同上
是	是	是	同上

消息可靠性

RabbitMQ消息的可靠性投递主要两种实现：
1、通过实现消费的重试机制，通过@Retryable来实现重试，可以设置重试次数和重试频率；
2、生产端实现消息可靠性投递。

两种方法消费端都可能收到重复消息，要求消费端必须实现幂等性消费。



消息的可靠投递

生产端

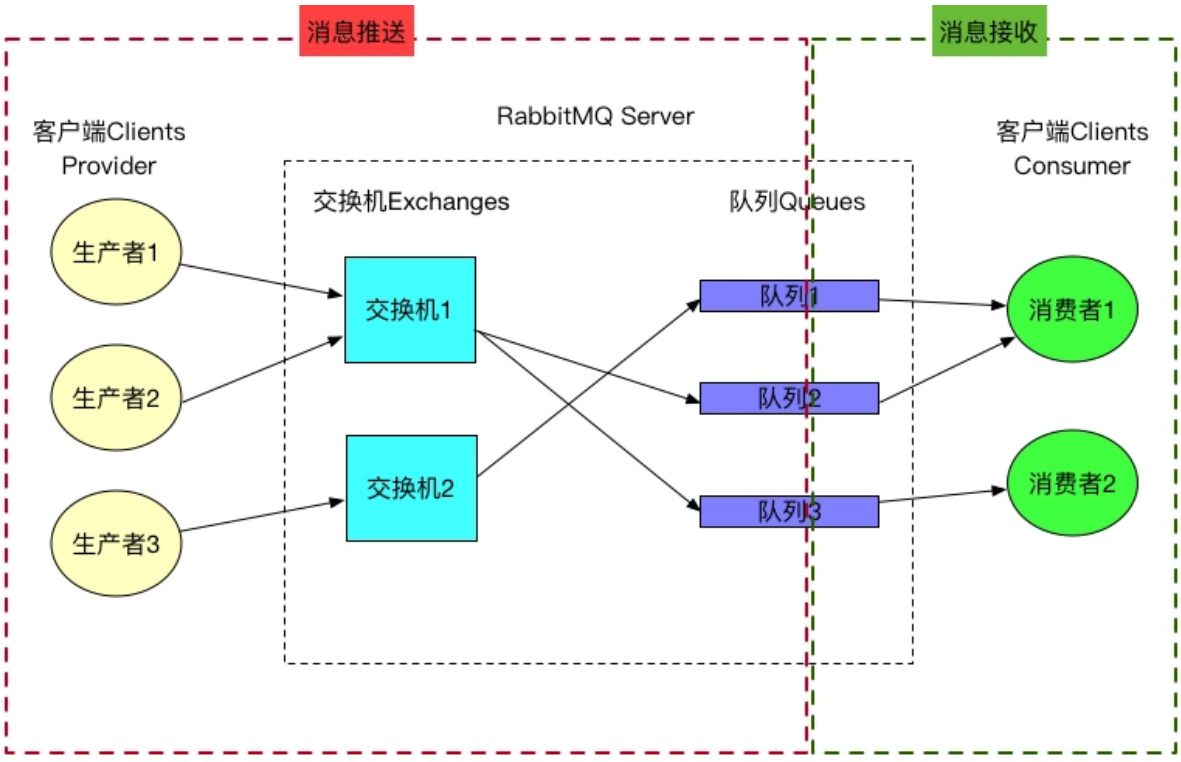
在使用 RabbitMQ 的时候，作为消息发送方希望杜绝任何消息丢失或者投递失败场景。RabbitMQ 为我们提供了两种方式用来控制消息的投递可靠性模式

- confirm 确认模式
- return 退回模式

消息投递到exchange的确认模式

rabbitmq的消息投递的过程为：

producer ——> rabbitmq broker cluster ——> exchange ——> queue ——> consumer



- 生产端发送消息到rabbitmq broker cluster后，异步接受从rabbitmq返回的ack确认信息
- 生产端收到返回的ack确认消息后，根据ack是true还是false，调用confirmCallback接口进行处理

1、**改yml**

```yaml
spring:
  #rabbitmq 连接配置
  rabbitmq:
    publisher-confirm-type: correlated # 开启confirm确认模式
```

2、**实现confirm方法**

实现ConfirmCallback接口中的confirm方法，消息只要被 rabbitmq broker接收到就会触ConfirmCallback 回调，ack为true表示消息发送成功，ack为false表示消息发送失败

```java
package com.rabbitmq.config;

import org.springframework.amqp.rabbit.connection.CorrelationData;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.stereotype.Component;

/**
 * 实现ConfirmCallback接口
 */
   @Component
public class ConfirmCallbackService implements
RabbitTemplate.ConfirmCallback {

    /**
     * @param correlationData 相关配置信息
     * @param ack exchange交换机 是否成功收到了消息。true 成功，false代表失败
     * @param cause 失败原因
     */
     @Override
     public void confirm(CorrelationData correlationData, boolean ack,
String cause) {
        if (ack) {
            //接收成功
            System.out.println("成功发送到交换机<===>");
        } else {
            //接收失败
            System.out.println("失败原因:===>" + cause);

            //TODO 做一些处理:消息再次发送等等

        }
     }
}
```

3、**测试**

定义 Exchange 和 Queue

定义交换机 confirmTestExchange 和队列 confirm_test_queue ， 并将队列绑定在交换机上。

```java
package com.rabbitmq.config;

import org.springframework.amqp.core.Binding;
import org.springframework.amqp.core.BindingBuilder;
import org.springframework.amqp.core.FanoutExchange;
import org.springframework.amqp.core.Queue;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.context.annotation.Bean;
```

```java
import org.springframework.context.annotation.Configuration;

/**
 * 队列与交换机绑定
 */
@Configuration
public class QueueConfig {

    @Bean(name = "confirmTestQueue")
    public Queue confirmTestQueue() {
        return new Queue("confirm_test_queue", true, false, false);
    }

    @Bean(name = "confirmTestExchange")
    public FanoutExchange confirmTestExchange() {
        return new FanoutExchange("confirmTestExchange");
    }

    @Bean
    public Binding confirmTestFanoutExchangeAndQueue(
            @Qualifier("confirmTestExchange") FanoutExchange confirmTestExchange,
            @Qualifier("confirmTestQueue") Queue confirmTestQueue) {
        return BindingBuilder.bind(confirmTestQueue).to(confirmTestExchange);
    }
}
```

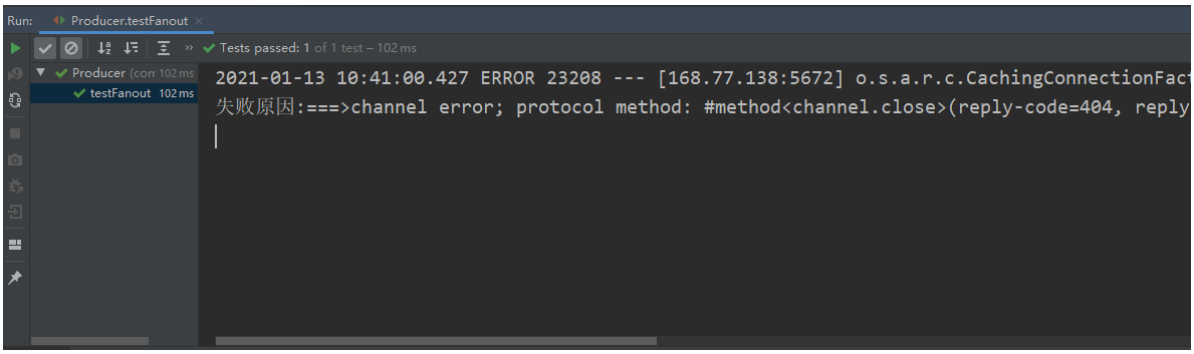
生产者

```java
@RunWith(SpringRunner.class)
@SpringBootTest(classes = RabbitmqApplication.class)
public class Producer {

    @Autowired
    private RabbitTemplate rabbitTemplate;  //注入rabbitmq对象

    @Autowired
    private ConfirmCallbackService confirmCallbackService; //注入ConfirmCallback对象

    @Test
    public void test() {
        //
        rabbitTemplate.setConfirmCallback(confirmCallbackService);
        //发送消息
        rabbitTemplate.convertAndSend("confirmTestExchange1", "", "hello,ConfirmCallback你好");
    }
}
```

正确情况，ack返回true，表示投递成功。

现在我们改变交换机名字，发送到一个不存在的交换机

```
//发送消息
rabbitTemplate.convertAndSend("confirmTestExchange1", "",
"hello,ConfirmCallback你好");
```



消息未投递到queue的退回模式

消息从 exchange–>queue 投递失败则会返回一个 returnCallback

生产端通过实现ReturnCallback接口，启动消息失败返回，消息路由不到队列时会触发该回调接口

1、**改yml**

```
spring:
  # rabbitmq 连接配置
  rabbitmq:
    publisher-returns: true # 开启退回模式
```

2、**设置投递失败的模式**

如果消息没有路由到Queue，则丢弃消息（默认）

如果消息没有路由到Queue，返回给消息发送方ReturnCallBack（开启后）

```
rabbitTemplate.setMandatory(true);
```

3、**实现returnedMessage方法**

启动消息失败返回，消息路由不到队列时会触发该回调接口

```
package com.rabbitmq.config;

import org.springframework.amqp.core.Message;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.stereotype.Component;

@Component
public class ReturnCallbackService implements RabbitTemplate.ReturnCallback
{

    /**
     * @param message    消息对象
     * @param replyCode  错误码
     * @param replyText  错误信息
     * @param exchange   交换机
     * @param routingKey 路由键
     */
     @Override
```

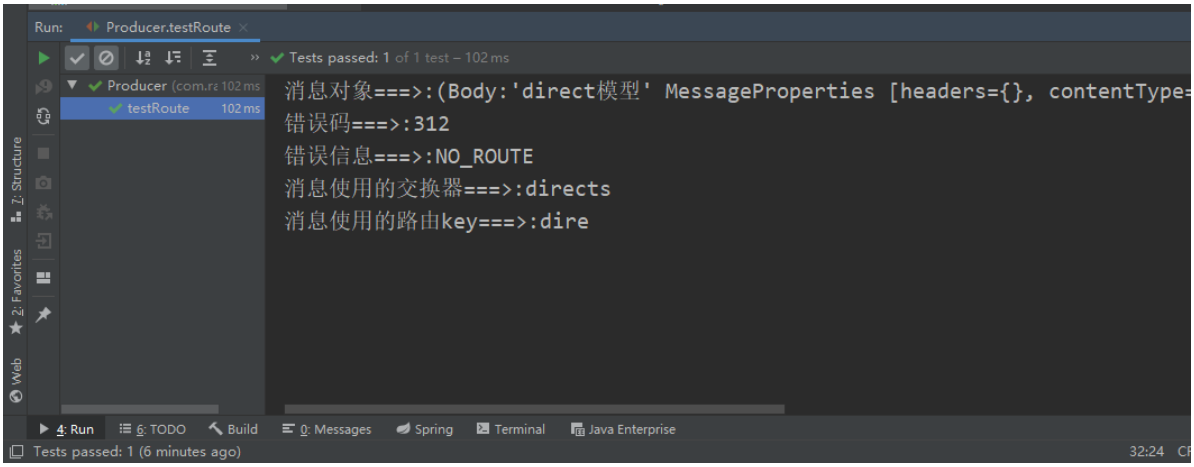
```
    public void returnedMessage(Message message, int replyCode, String replyText, String exchange, String routingKey) {

        System.out.println("消息对象===>:" + message);
        System.out.println("错误码===>:" + replyCode);
        System.out.println("错误信息===>:" + replyText);
        System.out.println("消息使用的交换器===>:" + exchange);
        System.out.println("消息使用的路由key===>:" + routingKey);

        //TODO ===>做业务处理
    }
}
```

4、**测试**

生产者

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes = RabbitmqApplication.class)
public class Producer {

    @Autowired
    private RabbitTemplate rabbitTemplate;  //注入rabbitmq对象
    @Autowired
    private ConfirmCallbackService confirmCallbackService;
    @Autowired
    private ReturnCallbackService returnCallbackService;

    @Test
    public void test() {

        /**
         * 确保消息发送失败后可以重新返回到队列中
         */
        rabbitTemplate.setMandatory(true);

        /**
         * 消息投递到队列失败回调处理
         */
        rabbitTemplate.setReturnCallback(returnCallbackService);

        /**
         * 消息投递确认模式
         */
        rabbitTemplate.setConfirmCallback(confirmCallbackService);

        //发送消息
        rabbitTemplate.convertAndSend("confirmTestExchange", "info", "hello,ConfirmCallback你好");
    }
}
```

如果不存在路由key"dire"，会调用ReturnCallback接口



消费端

消息确认机制ack

ack指Acknowledge确认。 表示消费端收到消息后的确认方式

消费端消息的确认分为：自动确认（默认）、手动确认、不确认

- AcknowledgeMode.NONE：不确认
- AcknowledgeMode.AUTO：自动确认
- AcknowledgeMode.MANUAL：手动确认

其中自动确认是指，当消息一旦被Consumer接收到，则自动确认收到，并将相应 message 从 RabbitMQ 的消息 缓存中移除。

但是在实际业务处理中，很可能消息接收到，业务处理出现异常，那么该消息就会丢失。如果设置了 手动确认方式，则需要在业务处理成功后，调用channel.basicAck()，手动签收，如果出现异常，则调用channel.basicNack()方法，让其自动重新发送消息。

1、**改yml**

```yaml
spring:
  rabbitmq:
    listener:
      simple:
        acknowledge-mode: manual # 手动确认
```

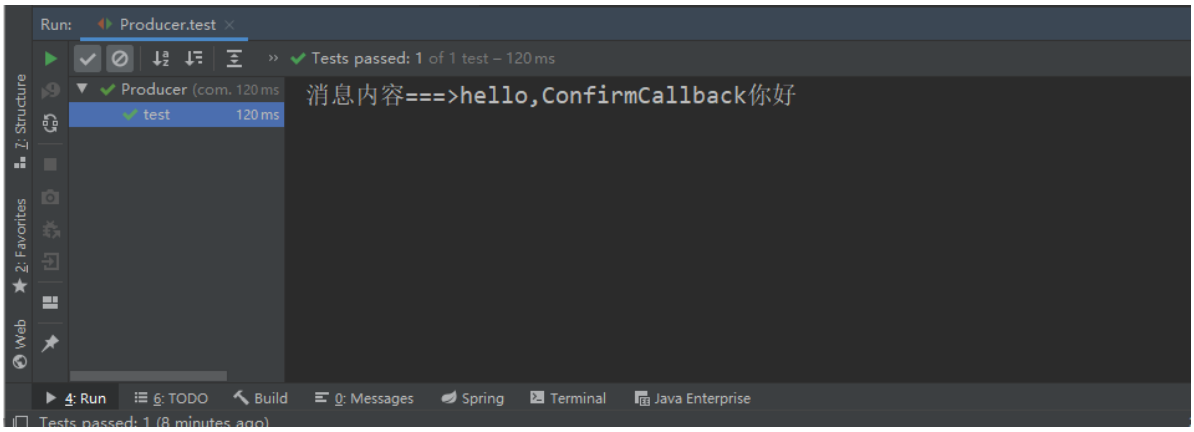
2、**确认配置**

```
@Component
@RabbitListener(queues = "confirm_test_queue")
public class ReceiverMessage {

    @RabbitHandler
    public void processHandler(String msg, Channel channel, Message message)
throws IOException {

        long deliveryTag = message.getMessageProperties().getDeliveryTag();
        try {

            System.out.println("消息内容===>" + new
String(message.getBody()));

```

```
            //TODO 具体业务逻辑

            //手动签收[参数1:消息投递序号,参数2:批量签收]
            channel.basicAck(deliveryTag, true);
        } catch (Exception e) {
            //拒绝签收[参数1:消息投递序号,参数2:批量拒绝,参数3:是否重新加入队列]
            channel.basicNack(deliveryTag, true, true);
        }
    }
}
```

channel.basicNack 方法与 channel.basicReject 方法区别在于basicNack可以批量拒绝多条消息，而basicReject一次只能拒绝一条消息。

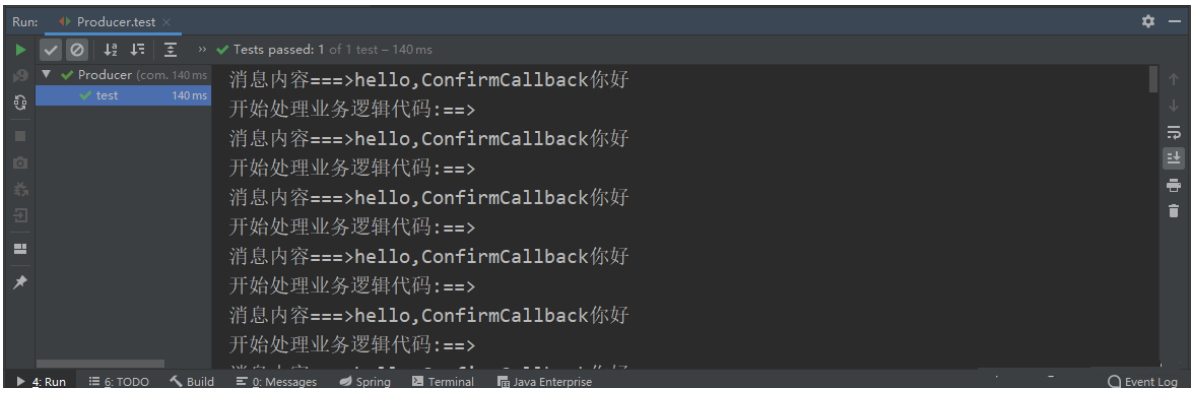
3、**测试**
正常情况



异常情况

在业务处理模块增加异常

```
//TODO 具体业务逻辑
System.out.println("开始处理业务逻辑代码:==>");
int i = 3/0;
```

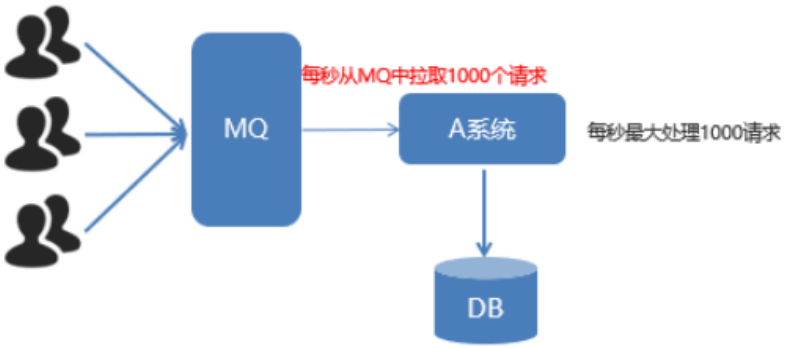


发生异常，拒绝确认，重新加入队列，一直循环，知道确认消息。

消费端限流

假设一个场景，首先，我们 Rabbitmq 服务器积压了有上万条未处理的消息，我们随便打开一个消费者客户端，会出现这样情况: 巨量的消息瞬间全部推送过来，但是我们单个客户端无法同时处理这么多数据!

当数据量特别大的时候，我们对生产端限流肯定是不科学的，因为有时候并发量就是特别大，有时候并发量又特别少，我们无法约束生产端，这是用户的行为。所以我们应该对消费端限流，用于保持消费端的稳定，当消息数量激增的时候很有可能造成资源耗尽，以及影响服务的性能，导致系统的卡顿甚至直接崩溃。



TTL

Time To Live，消息过期时间设置

声明队列时，指定即可

TTL:过期时间

1. 队列统一过期
2. 消息单独过期

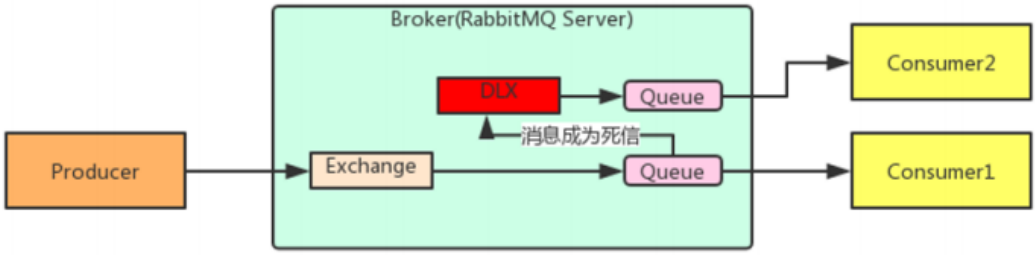
如果设置了消息的过期时间，也设置了队列的过期时间，它以时间短的为准。

- 队列过期后，会将队列所有消息全部移除
- 消息过期后，只有消息在队列顶端，才会判断其是否过期(移除掉)

死信队列

死信队列，英文缩写：DLX 。Dead Letter Exchange（死信交换机），当消息成为Dead message后，可以

被重新发送到另一个交换机，这个交换机就是DLX



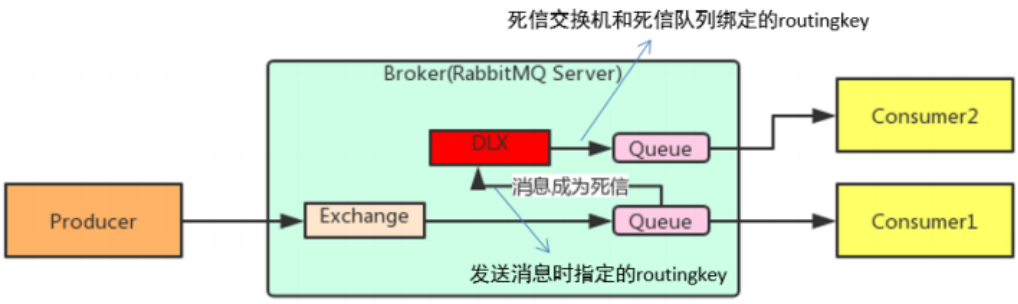
消息成为死信的三种情况：

1. 队列消息长度到达限制；
2. 消费者拒接消费消息，basicNack/basicReject,并且不把消息重新放入原目标队列,requeue=false；
3. 原队列存在消息过期设置，消息到达超时时间未被消费；

队列绑定死信交换机：

给队列设置参数： x-dead-letter-exchange 和 x-dead-letter-routing-key

也就是说此时Queue作为"生产者"



延迟队列

延迟队列，即消息进入队列后不会立即被消费，只有到达指定时间后，才会被消费

需求：

下单后，30分钟未支付，取消订单，回滚库存。

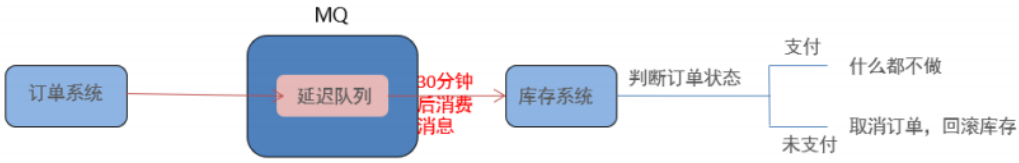
新用户注册成功7天后，发送短信问候。

实现方式：

定时器 （×）

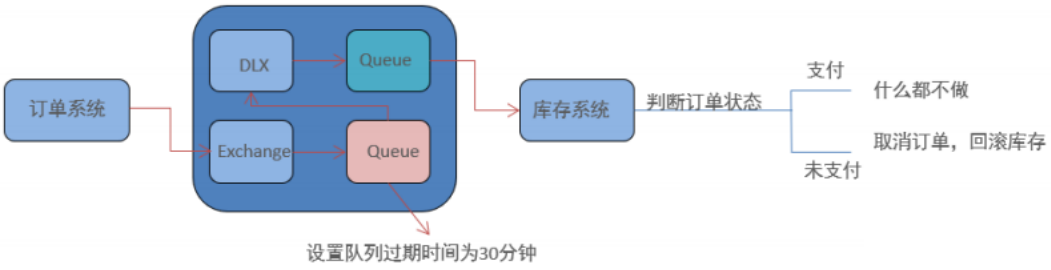
延迟队列 （√）

实现步骤：



在RabbitMQ中并未提供延迟队列功能

替代实现： TTL+死信队列 组合实现延迟队列的效果



设置队列过期时间30分钟，当30分钟过后，消息未被消费，进入死信队列，路由到指定队列，调用库存系统，判断订单状态。

rabbitmq监控

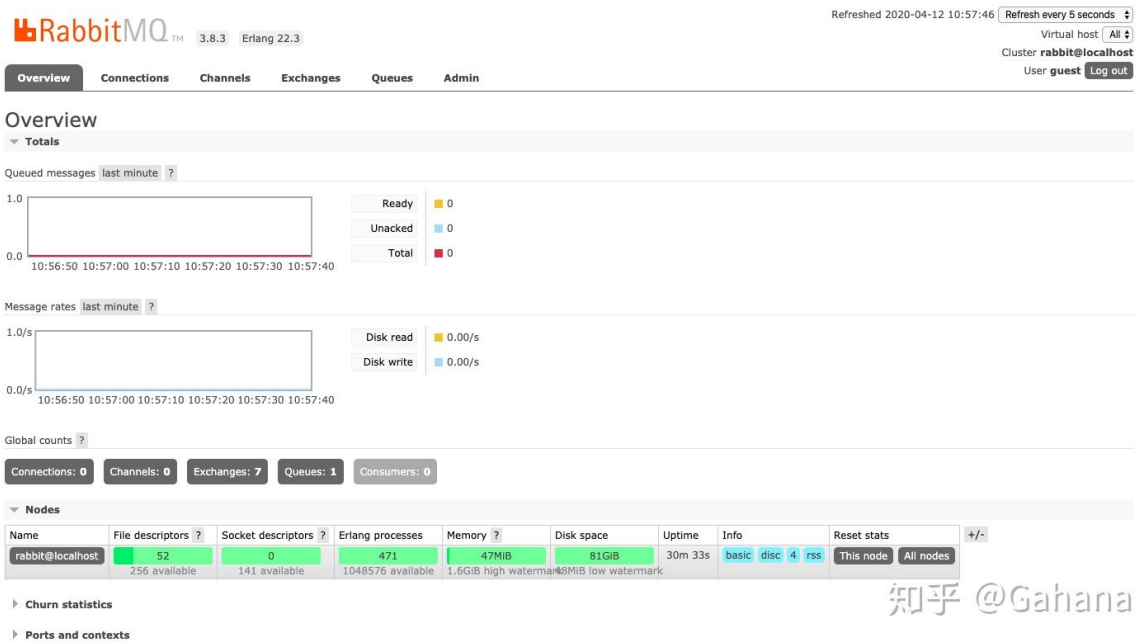
接下来说说监控的相关内容。

监控还是非常重要的，特别是在生产环境。磁盘满了，队列积压严重，如果我们还不知道，老板肯定会怀疑，莫不是这家伙要跑路？

而且我现在就遇到了这样的情况，主要是队列积压的问题。由于量不是很大，所以磁盘空间倒不是很担心，但有时程序执行会报错，导致队列一直消费不下去，这就很让人尴尬了。

查了一些资料，总结了一下。想要了解 RabbitMQ 的运行状态，主要有三种途径：Management UI，rabbitmqctl 命令和 REST API 以及使用 prometheus + grafana，当然大厂的话都会自定开发基于 api 监控系统。

Management UI



RabbitMQ 给我们提供了丰富的 Web 管理功能，通过页面，我们能看到 RabbitMQ 的整体运行状况，交换机和队列的状态等，还可以进行人员管理和权限配置，相当全面。

但如果想通过页面来监控，那出不出问题只能靠缘分。看到出问题了，是运气好，看不到出问题，那是必然。

这也是我当前的现状，所以为了避免出现大问题，得赶紧改变一下。

备注：通过 http://127.0.0.1:15672 来访问 Web 页面，默认情况下用户名和密码都是 guest，但生产环境下都应该改掉的。

rabbitmqctl 命令

与前端页面对应的就是后端的命令行命令了，同样非常丰富。平时自己测试，或者临时查看一些状态时，也能用得上。但就我个人使用感觉来说，用的并不是很多。

我总结一些还算常用的，列在下面，大家各取所需：

```
# 启动服务
rabbitmq-server

# 停止服务
rabbitmqctl stop

# vhost 增删查
rabbitmqctl add_vhost
rabbitmqctl delete_vhost
rabbitmqctl list_vhosts

# 查询交换机
rabbitmqctl list_exchanges

# 查询队列
rabbitmqctl list_queues

# 查看消费者信息
rabbitmqctl list_consumers

# user 增删查
rabbitmqctl add_user
rabbitmqctl delete_user
rabbitmqctl list_users
```

REST API

终于来到重点了，对于程序员来说，看到有现成的 API 可以调用，那真是太幸福了。

自动化监控和一些需要批量的操作，通过调用 API 来实现是最好的方式。比如有一些需要初始化的用户和权限，就可以通过脚本来一键完成，而不是通过页面逐个添加，简单又快捷。

下面是一些常用的 API：

```
# 概括信息
curl -i -u guest:guest http://localhost:15672/api/overview
```

```
# vhost 列表
curl -i -u guest:guest http://localhost:15672/api/vhosts

# channel 列表
curl -i -u guest:guest http://localhost:15672/api/channels

# 节点信息
curl -i -u guest:guest http://localhost:15672/api/nodes

# 交换机信息
curl -i -u guest:guest http://localhost:15672/api/exchanges

# 队列信息
curl -i -u guest:guest http://localhost:15672/api/queues
```

就我现在遇到的情况来说，`overview` 和 `queues` 这两个 API 就可以满足我的需求，大家也可以根据自己项目的实际情况来选择。

API 返回内容是 json，而且字段还是挺多的，刚开始看会感觉一脸懵，具体含义对照官网的解释和实际情况来慢慢琢磨，弄懂也不是很困难。

下面是一个demo代码，主要使用HttpClient以及jackson来调用相关参数。
相关maven如下：

```
<dependency>
    <groupId>org.apache.httpcomponents</groupId>
    <artifactId>httpclient</artifactId>
    <version>4.3.6</version>
</dependency>
<dependency>
 <groupId>com.fasterxml.jackson.core</groupId>
 <artifactId>jackson-databind</artifactId>
 <version>2.7.4</version>
</dependency>
<dependency>
 <groupId>com.fasterxml.jackson.core</groupId>
 <artifactId>jackson-annotations</artifactId>
 <version>2.7.4</version>
</dependency>
<dependency>
 <groupId>com.fasterxml.jackson.core</groupId>
 <artifactId>jackson-core</artifactId>
 <version>2.7.4</version>
</dependency>
```

```
impor
```

相关代码（有点长）：

```
package com.vms.test.zzh.rabbitmq.monitor;

import com.fasterxml.jackson.databind.DeserializationFeature;
import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.SerializationFeature;
```

```
7
8  import org.apache.http.HttpEntity;
9  import org.apache.http.auth.UsernamePasswordCredentials;
10 import org.apache.http.client.methods.CloseableHttpResponse;
11 import org.apache.http.client.methods.HttpGet;
12 import org.apache.http.impl.auth.BasicScheme;
13 import org.apache.http.impl.client.CloseableHttpClient;
14 import org.apache.http.impl.client.HttpClients;
15 import org.apache.http.util.EntityUtils;
16
17 import java.io.IOException;
18 import java.util.HashMap;
19 import java.util.Iterator;
20 import java.util.Map;
21
22 public class MonitorDemo {
23     public static void main(String[] args) {
24         try {
25             Map<String, ClusterStatus> map =
fetchNodesStatusData("http://localhsot:15672/api/nodes", "root", "root");
26             for (Map.Entry entry : map.entrySet()) {
27                 System.out.println(entry.getKey() + " : " +
entry.getValue());
28             }
29         } catch (IOException e) {
30             e.printStackTrace();
31         }
32     }
33
34     public static Map<String,ClusterStatus> fetchNodesStatusData(String
url, String username, String password) throws IOException {
35         Map<String, ClusterStatus> clusterStatusMap = new HashMap<String,
ClusterStatus>();
36         String nodeData = getData(url, username, password);
37         JsonNode jsonNode = null;
38         try {
39             jsonNode = JsonUtil.toJsonNode(nodeData);
40         } catch (IOException e) {
41             e.printStackTrace();
42         }
43         Iterator<JsonNode> iterator = jsonNode.iterator();
44         while (iterator.hasNext()) {
45             JsonNode next = iterator.next();
46             ClusterStatus status = new ClusterStatus();
47             status.setDiskFree(next.get("disk_free").asLong());
48             status.setFdUsed(next.get("fd_used").asLong());
49             status.setMemoryUsed(next.get("mem_used").asLong());
50             status.setProcUsed(next.get("proc_used").asLong());
51             status.setSocketUsed(next.get("sockets_used").asLong());
52             clusterStatusMap.put(next.get("name").asText(), status);
53         }
54         return clusterStatusMap;
55     }
56
57     public static String getData(String url, String username, String
password) throws IOException {
58         CloseableHttpClient httpClient = HttpClients.createDefault();
```

```
59          UsernamePasswordCredentials creds = new
UsernamePasswordCredentials(username, password);
60          HttpGet httpGet = new HttpGet(url);
61          httpGet.addHeader(BasicScheme.authenticate(creds, "UTF-8", false));
62          httpGet.setHeader("Content-Type", "application/json");
63          CloseableHttpResponse response = httpClient.execute(httpGet);
64
65          try {
66              if (response.getStatusLine().getStatusCode() != 200) {
67                  System.out.println("call http api to get rabbitmq data
return code: " + response.getStatusLine().getStatusCode() + ", url: " +
url);
68              }
69              HttpEntity entity = response.getEntity();
70              if (entity != null) {
71                  return EntityUtils.toString(entity);
72              }
73          } finally {
74              response.close();
75          }
76
77          return "";
78      }
79
80      public static class JsonUtil {
81          private static ObjectMapper objectMapper = new ObjectMapper();
82          static {
83
 objectMapper.disable(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES);
84              objectMapper.disable(SerializationFeature.FAIL_ON_EMPTY_BEANS);
85          }
86
87          public static JsonNode toJsonNode(String jsonString) throws
IOException {
88              return objectMapper.readTree(jsonString);
89          }
90      }
91
92      public static class ClusterStatus {
93          private long diskFree;
94          private long diskLimit;
95          private long fdUsed;
96          private long fdTotal;
97          private long socketUsed;
98          private long socketTotal;
99          private long memoryUsed;
100         private long memoryLimit;
101         private long procUsed;
102         private long procTotal;
103         // 此处省略了Getter和Setter方法
104         @Override
105         public String toString() {
106             return "ClusterStatus{" +
107                     "diskFree=" + diskFree +
108                     ", diskLimit=" + diskLimit +
109                     ", fdUsed=" + fdUsed +
110                     ", fdTotal=" + fdTotal +
111                     ", socketUsed=" + socketUsed +
```

```
                    ", socketTotal=" + socketTotal +
                    ", memoryUsed=" + memoryUsed +
                    ", memoryLimit=" + memoryLimit +
                    ", procUsed=" + procUsed +
                    ", procTotal=" + procTotal +
                    '}';
        }
    }
}
```

代码输出：

```
rabbit@zhuzhonghua2-fqawb : ClusterStatus{diskFree=34480971776, diskLimit=0, fdUsed=38, fdTotal=0, socketUsed=1, socketTotal=0, memoryUsed=44508400, memoryLimit=0, procUsed=196, procTotal=0}
rabbit@hiddenzhu-8drdc : ClusterStatus{diskFree=36540743680, diskLimit=0, fdUsed=28, fdTotal=0, socketUsed=1, socketTotal=0, memoryUsed=53331640, memoryLimit=0, procUsed=181, procTotal=0}
```

通过对返回结果进行解析，就可以判断 RabbitMQ 的整体运行状态，如果发生超阈值的情况，可以发送告警或邮件，来达到监控的效果。

针对队列积压情况的监控判断，有两种方式：

- 一是设置队列积压长度阈值，如果超过阈值即告警；
- 二是保存最近五次的积压长度，如果积压逐渐增长并超阈值，即告警。

第二种方式更好，判断更加精准，误告可能性小，但实现起来也更复杂。

这里只是提一个思路，等后续再把实践结果和代码分享出来。或者大家有哪些更好的方法吗？欢迎留言交流。

prometheus + grafana 监控rabbitmq