

RabbitMQ研究

|  |  |
| --- | --- |
| 文件编号： |  |
| 文件版本： |  |
| **拟 制：** | 张颂扬 2020年 11月 6日 |
| **审 核：** | 年 月 日 |
| **批 准：** | 年 月 日 |

目录

[1 快速入门 5](#_Toc54950259)

[1.1 平台结构 5](#_Toc54950260)

[1.1.1 主菜单 5](#_Toc54950261)

[1.1.2 开发总流程 5](#_Toc54950262)

[1.1.3 帮助文档 6](#_Toc54950263)

[1.2 重要提醒 7](#_Toc54950264)

[1.2.1 测试机器 7](#_Toc54950265)

[1.2.2 预发布/正式发布 7](#_Toc54950266)

[1.2.3 不建议使用字典 7](#_Toc54950267)

[1.2.4 模型字段 7](#_Toc54950268)

[1.2.5 流程、非流程 8](#_Toc54950269)

[1.2.6 帮助文档入口 8](#_Toc54950270)

[1.3 数据库及服务器 9](#_Toc54950271)

[1.3.1 数据库 9](#_Toc54950272)

[1.3.2 服务器 13](#_Toc54950273)

[2 数据建模 13](#_Toc54950274)

[2.1 开发总流程 13](#_Toc54950275)

[2.2 模型开发流程 14](#_Toc54950276)

[2.3 功能入口 14](#_Toc54950277)

[2.4 新建模型 15](#_Toc54950278)

[2.4.1 模型编码 16](#_Toc54950279)

[2.4.2 所属系统 16](#_Toc54950280)

[2.4.3 服务名 16](#_Toc54950281)

[2.4.4 客户/领域/子领域 17](#_Toc54950282)

[2.4.5 视图SQL语句 17](#_Toc54950283)

[2.4.6 基础属性其它字段 17](#_Toc54950284)

[2.4.7 模型属性字段定义 18](#_Toc54950285)

[2.4.8 模型方法定义 22](#_Toc54950286)

[2.4.9 模型属性查询规则 22](#_Toc54950287)

[2.4.10 模型属性导入规则 23](#_Toc54950288)

[2.5 实例-非工作流模型/工作流模型 25](#_Toc54950289)

[2.6 实例-接口模型 27](#_Toc54950290)

[3 组件定义 28](#_Toc54950291)

[3.1 新建组件 29](#_Toc54950292)

[3.2 组件属性 30](#_Toc54950293)

[3.3 组件的设置-表格 33](#_Toc54950294)

[3.3.1 组件属性 33](#_Toc54950295)

[3.3.2 表格属性 34](#_Toc54950296)

[3.3.3 表格属性-格式化 35](#_Toc54950297)

[3.4 组件的设置-表单 36](#_Toc54950298)

[3.5 组件-弹出窗口 38](#_Toc54950299)

[4 界面定义 40](#_Toc54950300)

[4.1 新建界面 40](#_Toc54950301)

[4.2 界面属性 41](#_Toc54950302)

[4.3 界面使用 42](#_Toc54950303)

[4.4 界面联动 42](#_Toc54950304)

[5 字典定义（不建议使用） 43](#_Toc54950305)

[5.1 新建字典 43](#_Toc54950306)

[5.2 字典使用 45](#_Toc54950307)

[5.3 预发布/正式发布 45](#_Toc54950308)

[6 方法定义 46](#_Toc54950309)

[6.1 新建方法 46](#_Toc54950310)

[6.2 预发布/正式发布 47](#_Toc54950311)

[7 审批流程 47](#_Toc54950312)

[7.1 工作流模型 47](#_Toc54950313)

[7.2 业务组件 47](#_Toc54950314)

[7.3 流程组件 48](#_Toc54950315)

[7.4 流程定义 49](#_Toc54950316)

[7.5 测试流程 51](#_Toc54950317)

[7.6 审批流程帮助文档 51](#_Toc54950318)

[8 定时任务 52](#_Toc54950319)

[8.1 新建定时任务 52](#_Toc54950320)

[8.2 定时任务后台代码 54](#_Toc54950321)

[9 邮件发送 56](#_Toc54950322)

[9.1 发送邮件 56](#_Toc54950323)

[10 应用管理 57](#_Toc54950324)

[10.1 管理入口 57](#_Toc54950325)

[10.2 建立应用 57](#_Toc54950326)

[10.3 应用使用 60](#_Toc54950327)

[11 调试技巧 60](#_Toc54950328)

[11.1 调试环境 60](#_Toc54950329)

[11.2 配置调试端口 64](#_Toc54950330)

# 概述

## MQ

消息总线(Message Queue)，是一种跨进程、异步的通信机制，用于上下游传递消息。**由消息系统来确保消息的可靠传递。**

**作用：**应用解耦、异步、流量削锋、数据分发、错峰流控、日志收集等。

**MQ衡量标准：**服务性能、数据存储、集群架构

## 主流竞品分析

当前市面上mq的产品很多，比如RabbitMQ、Kafka、ActiveMQ、ZeroMQ和阿里巴巴捐献给Apache的RocketMQ。甚至连redis这种NoSQL都支持MQ的功能。

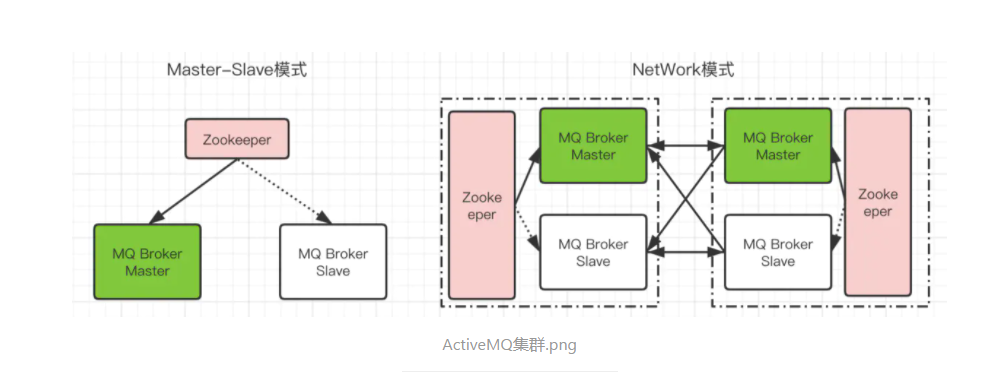
ActiveMQ

ActiveMQ是apache出品，最流行的，能力强劲的开源消息总线，并且它一个完全支持JMS规范的消息中间件。其丰富的API、多种集群构建模式使得它成为业界老牌消息中间件，在中小型企业中应用广泛。

但是其性能稍差，在面对高并发的情况下，会出现消息阻塞、堆积、延迟等问题。

默认采用了基于内存的kahaDB进行存储，如果需要保证消息的可靠性，也可以选择关系行数据库进行存储。

集群架构模式如下：



Master-Slave模式：通过zookeeper对主从进行管理，正常情况下，从节点不会提供服务。当主节点出现问题后，zookeeper会高效的将主节点下掉，从节点来提供服务。

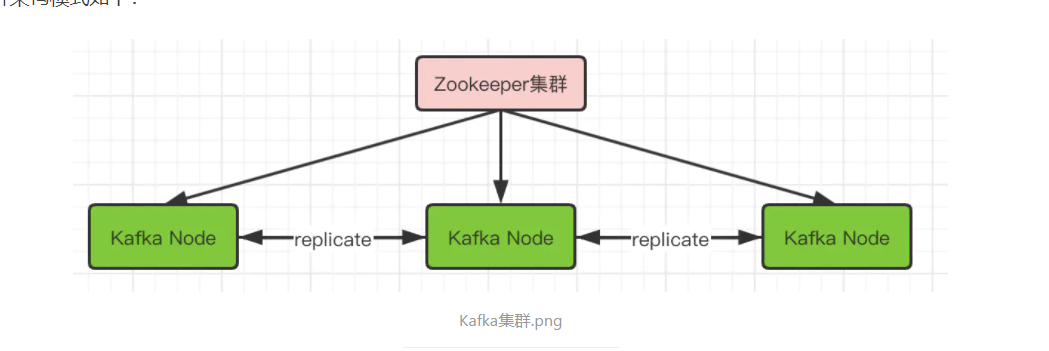
NetWork模式：两套主从Master-Slave节点。由网络联通，将其变为分布式的集群架构。

Kafka

Kafka是LinkedIn开源的分布式发布-订阅消息系统，目前归属于Apache顶级项目。Kafka主要特点就是**基于Pull的模式来处理消息消费**，追求高吞吐量，一开始的目的就是用于日志收集和传输。0.8版本开始支持复制，不支持事务，对消息的重复、丢失、错误没有严格要求，适合产生大量数据的互联网服务的数据收集业务。能够支持廉价的服务器上以每秒100k条数据的吞吐量。(有ack机制，可以保证不丢失，不能保证不重复。)

高效的读写基于操作系统低层的Page Cache。仅仅使用内存管理，不存在内存和磁盘之间的IO操作。

集群架构模式如下：



通过replicate进行节点间数据的复制，尽量保证数据的可靠性。

RocketMQ

RocketMQ是阿里开源的消息中间件，目前也已经孵化为Apache顶级项目，它是纯Java开发，具有高吞吐量、高可靠性、适合大规模分布式系统应用的特点。RocketMQ思路起源于Kafka，它对消息的可靠传输以及事务性做了优化，目前在阿里集团被广泛应用于交易、充值、流计算、消息推送、日志流式处理、binlog分发等场景。

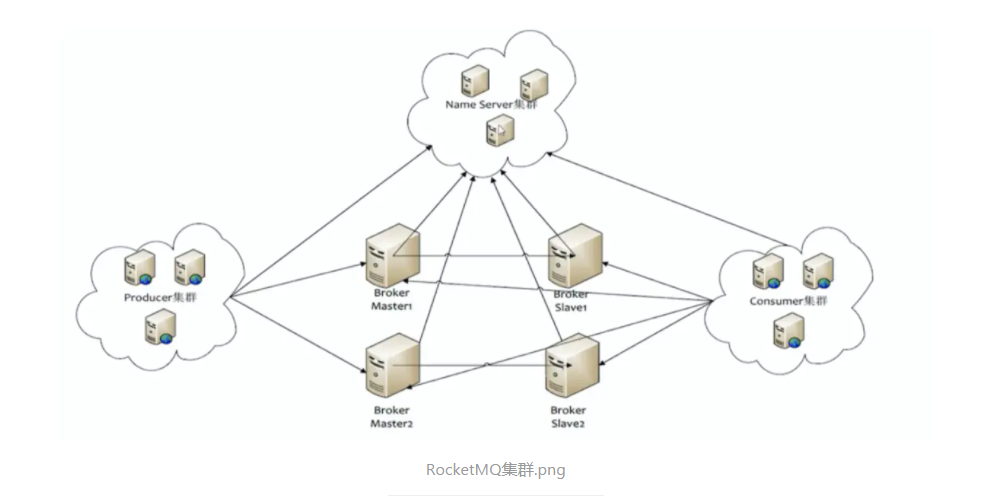
在2.0版本，RocketMQ集群也是通过Zookeeper进行管理。在3.0之后，放弃Zookeeper，使用NameServer进行集群的管理和协调。

能够保障消息的顺序消费，提供了丰富的消息拉取等处理模式，消费者可以高效进行水平扩展，能够承载上亿级别数据量级。

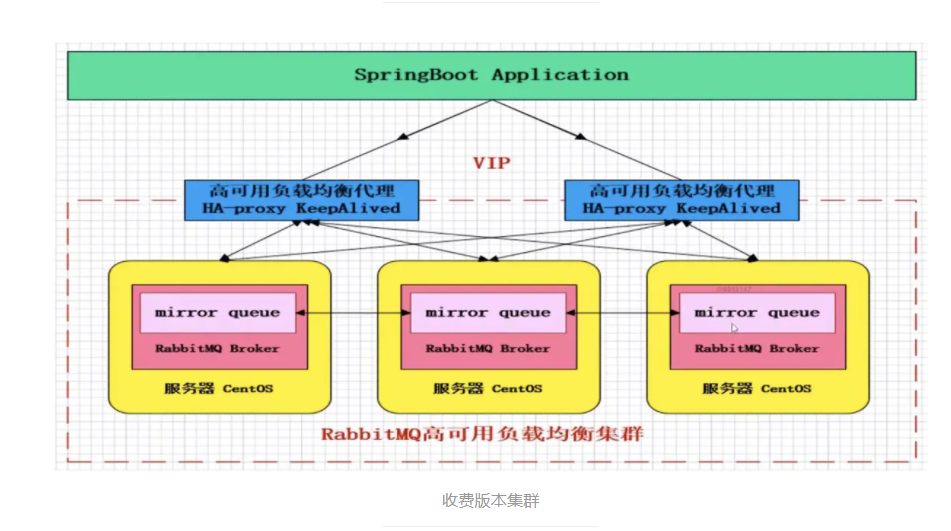
可以支持多种集群架构模式：Master-Slave模式、双Master-Slave模式、多主多从模式等等。

支持多种刷盘策略：同步双写、异步复制。借助了零拷贝等技术。

集群架构模式如下：



收费版本集群



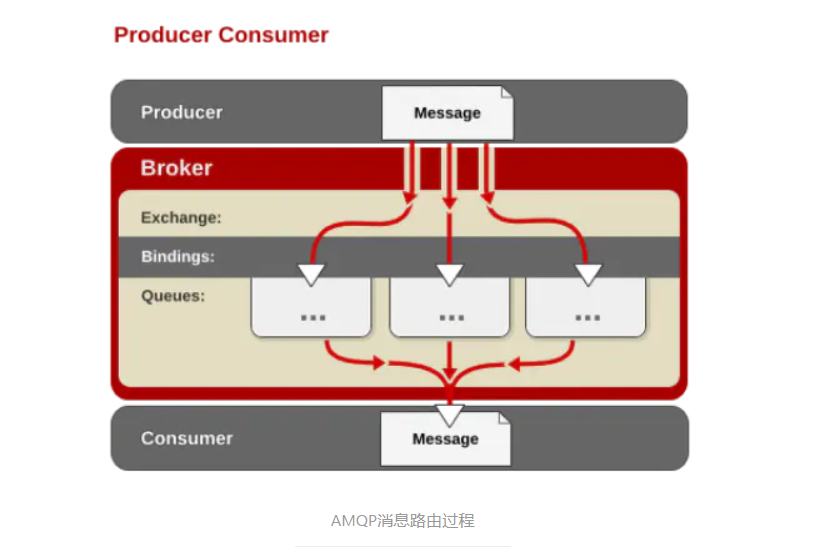


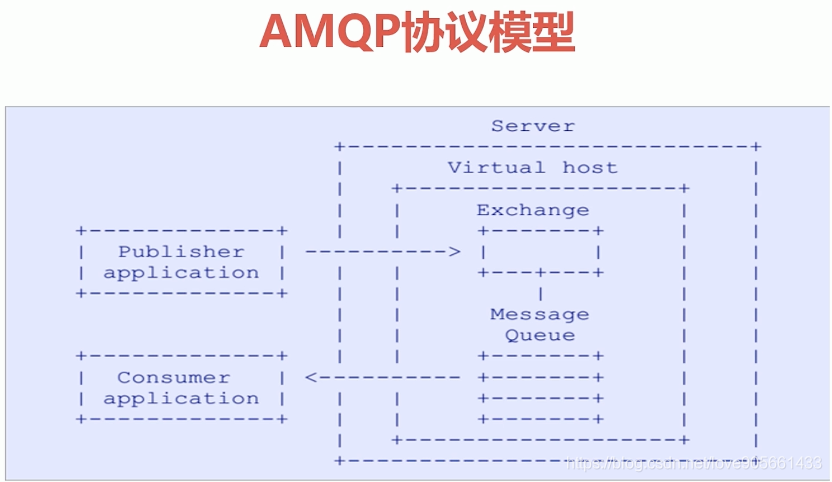
## 技术背景知识介绍

### AMQP高级消息队列协议

**AMQP(Advanced Message Queuing Protocol)**高级消息队列协议：高级消息队列协议。它是应用层协议的一个开放标准，为面向消息的中间件设计，基于此协议的客户端与消息中间件可传递消息，并不受产品、开发语言等条件的限制。

AMQP中消息的路由过程和JMS存在一些差别。**AMQP中增加了Exchange和Binging的角色。生产者把消息发布到Exchange上，消息最终到达队列并被消费者接收，而Binding决定交换器的消息应该发送到哪个队列。**





1. 生产者生产的消息通过Server->Virtual Host->Exchange
2. Exchange和Queue之间进行绑定
3. 消费者只需要监听Queue消息队列即可

### Erlang语言

**Erlang语言**最初用于交换机领域的架构模式，这样使得RabbitMQ在Broker之间进行数据交互的性能非常优秀(Erlang有着和原生Socket一样的延迟)。

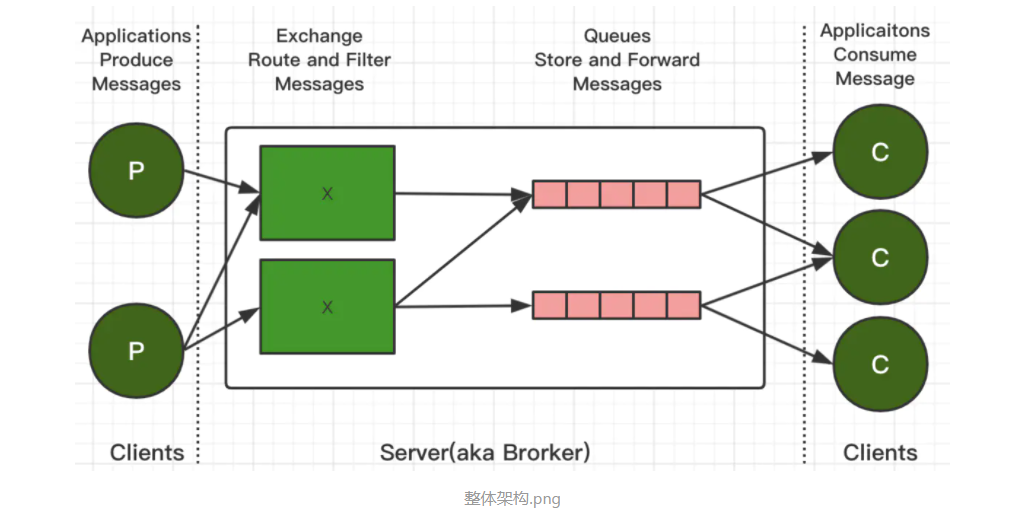
### RabbitMQ

RabbitMQ是一个开源的消息代理和队列服务器，用来通过普通协议在不同的应用之间共享数据(跨平台跨语言)。RabbitMQ是使用Erlang语言编写，并且基于AMQP协议实现。

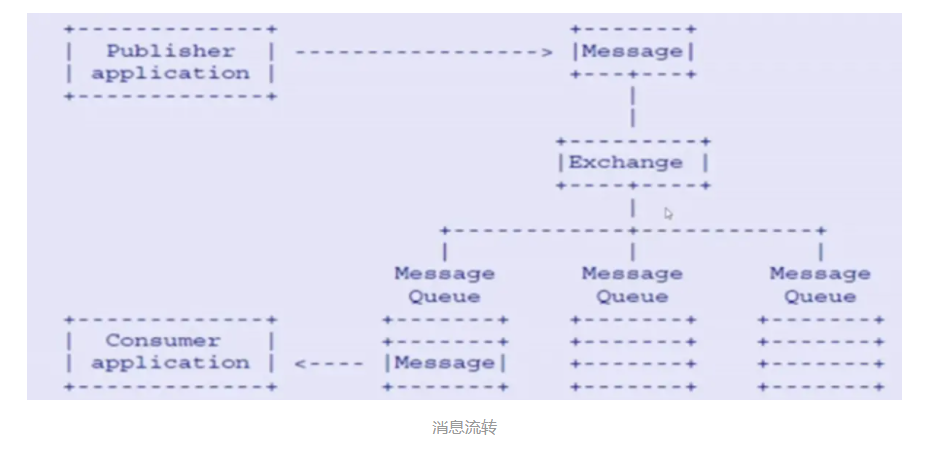
#### RabbitMQ的优势：

* **可靠性(Reliablity)：**使用了一些机制来保证可靠性，比如持久化、传输确认、发布确认。
* **灵活的路由(Flexible Routing)：**在消息进入队列之前，通过Exchange来路由消息。对于典型的路由功能，Rabbit已经提供了一些内置的Exchange来实现。针对更复杂的路由功能，可以将多个Exchange绑定在一起，也通过插件机制实现自己的Exchange。
* **消息集群(Clustering)：**多个RabbitMQ服务器可以组成一个集群，形成一个逻辑Broker。
* **高可用(Highly Avaliable Queues)：**队列可以在集群中的机器上进行镜像，使得在部分节点出问题的情况下队列仍然可用。
* **多种协议(Multi-protocol)：**支持多种消息队列协议，如STOMP、MQTT等。
* **多种语言客户端(Many Clients)：**几乎支持所有常用语言，比如Java、.NET、Ruby等。
* **管理界面(Management UI)：**提供了易用的用户界面，使得用户可以监控和管理消息Broker的许多方面。
* **跟踪机制(Tracing)：**如果消息异常，RabbitMQ提供了消息的跟踪机制，使用者可以找出发生了什么。
* **插件机制(Plugin System)：**提供了许多插件，来从多方面进行扩展，也可以编辑自己的插件。

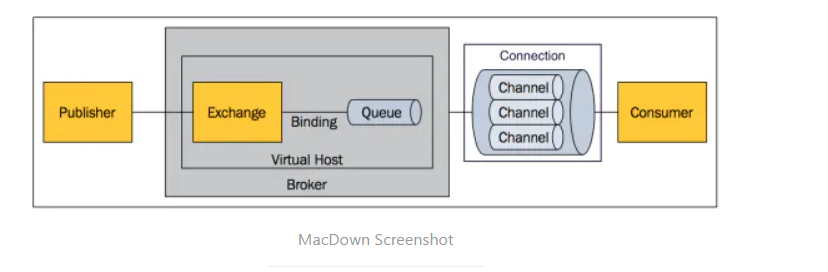
#### RabbitMQ的整体架构



#### RabbitMQ的消息流转



#### RabbitMQ各组件功能



 **Broker：**标识消息队列服务器实体.

 **Virtual Host：**虚拟主机。标识一批交换机、消息队列和相关对象。虚拟主机是共享相同的身份认证和加密环境的独立服务器域。每个vhost本质上就是一个mini版的RabbitMQ服务器，拥有自己的队列、交换器、绑定和权限机制。vhost是AMQP概念的基础，必须在链接时指定，RabbitMQ默认的vhost是 /。

 **Exchange：**交换器，用来接收生产者发送的消息并将这些消息路由给服务器中的队列。

 **Queue：**消息队列，用来保存消息直到发送给消费者。它是消息的容器，也是消息的终点。一个消息可投入一个或多个队列。消息一直在队列里面，等待消费者连接到这个队列将其取走。

 **Banding：**绑定，用于消息队列和交换机之间的关联。一个绑定就是基于路由键将交换机和消息队列连接起来的路由规则，所以可以将交换器理解成一个由绑定构成的路由表。

 **Channel：**信道，多路复用连接中的一条独立的双向数据流通道。信道是建立在真实的TCP连接内地虚拟链接，AMQP命令都是通过信道发出去的，不管是发布消息、订阅队列还是接收消息，这些动作都是通过信道完成。因为对于操作系统来说，建立和销毁TCP都是非常昂贵的开销，所以引入了信道的概念，以复用一条TCP连接。

 **Connection：**网络连接，比如一个TCP连接。

 **Publisher：**消息的生产者，也是一个向交换器发布消息的客户端应用程序。

 **Consumer：**消息的消费者，表示一个从一个消息队列中取得消息的客户端应用程序。

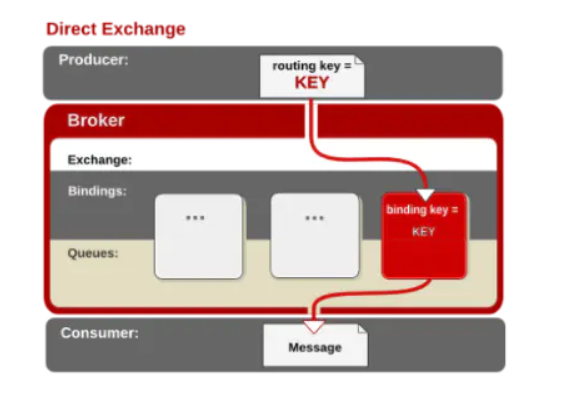
 **Message：**消息，消息是不具名的，它是由消息头和消息体组成。消息体是不透明的，而消息头则是由一系列的可选属性组成，这些属性包括routing-key(路由键)、priority(优先级)、delivery-mode(消息可能需要持久性存储[消息的路由模式])等。

#### RabbitMQ的多种Exchange类型

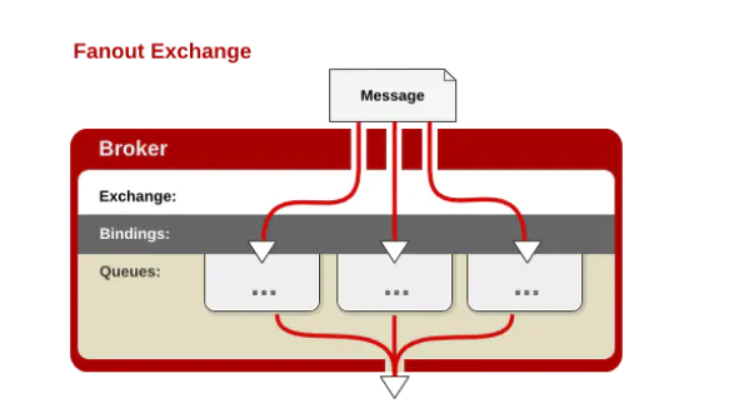
Exchange分发消息时，根据类型的不同分发策略有区别。目前共四种类型：direct、fanout、topic、headers(headers匹配AMQP消息的header而不是路由键(Routing-key)，此外headers交换器和direct交换器完全一致，但是性能差了很多，目前几乎用不到了。所以直接看另外三种类型。)。

direct:

消息中的路由键(routing key)如果和Binding中的binding key一致，交换器就将消息发到对应的队列中。路由键与队列名完全匹配。

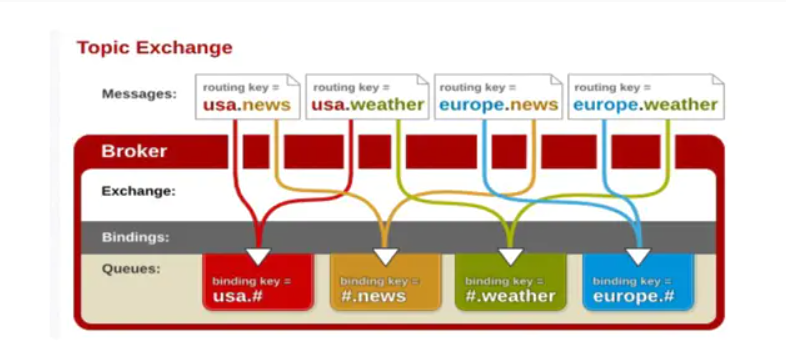


Fanout:



每个发到fanout类型交换器的消息都会分到所有绑定的队列上去。fanout交换器不处理该路由键，只是简单的将队列绑定到交换器上，每个发送到交换器的消息都会被转发到与该交换器绑定的所有队列上。很像子网广播，每台子网内的主机都获得了一份复制的消息。fanout类型转发消息是最快的。

topic:



topic交换器通过模式匹配分配消息的路由键属性，将路由键和某个模式进行匹配，此时队列需要绑定到一个模式上。它将路由键(routing-key)和绑定键(bingding-key)的字符串切分成单词，这些单词之间用点隔开。它同样也会识别两个通配符："#"和"\*"。#匹配0个或多个单词，匹配不多不少一个单词。

## TTL

TTL(Time To Live)：生存时间。RabbitMQ支持消息的过期时间，一共两种。

* 在消息发送时可以进行指定。通过配置消息体的properties，可以指定当前消息的过期时间。
* 在创建Exchange时可进行指定。从进入消息队列开始计算，只要超过了队列的超时时间配置，那么消息会自动清除

### TTL队列/消息

* RabbitMQ支持消息的过期时间, 在消息发送时可以进行指定
* RabbitMQ支持队列的过期时间, 从消息入队列开始计算, 只要超过了队列的超时时间配置, 那么消息会自动清除

### 代码演示

* 消费者中设置队列超时时间为10秒, 启动之后关闭消费者
* 生产者发送两条消息, 一条消息不设置超时时间, 一条消息设置5秒后超时
* 启动生产者之后, 监控RabbitMQ控制台, 发现5秒后设置了消息超时时间的消息先超时清除, 然后10秒后另外一条消息也超时清除

package com.qiyexue.api.ttl;

import com.rabbitmq.client.Channel;

import com.rabbitmq.client.Connection;

import com.rabbitmq.client.ConnectionFactory;

import com.rabbitmq.client.QueueingConsumer;

import com.rabbitmq.client.QueueingConsumer.Delivery;

import java.util.HashMap;

import java.util.Map;

/\*\*

\* 消费者， 设置队列过期时间

\*

\* @author 七夜雪

\* @date 2018-12-13 20:57

\*/

public class Consumer {

public static void main(String[] args) throws Exception {

// 1. 创建连接工厂, 设置属性

ConnectionFactory factory = new ConnectionFactory();

factory.setHost("192.168.72.138");

factory.setPort(5672);

factory.setVirtualHost("/");

// 2. 创建连接

Connection connection = factory.newConnection();

// 3. 使用connection创建channel

Channel channel = connection.createChannel();

// 4. 声明(创建)一个队列

String queueName = "test\_ttl\_queue";

Map<String, Object> arguments = new HashMap<>();

// 设置队列超时时间为10秒

arguments.put("x-message-ttl", 10000);

channel.queueDeclare(queueName,true, false, false, arguments);

// 5. 创建消费者

QueueingConsumer consumer = new QueueingConsumer(channel);

// 6. 设置channel

channel.basicConsume(queueName, true, consumer);

while (true) {

// 7. 获取消息

Delivery delivery = consumer.nextDelivery();

System.out.println(new String(delivery.getBody()));

// 获取head中内容

System.out.println(delivery.getProperties().getHeaders().get("name"));

}

}

}

package com.qiyexue.api.ttl;

import com.rabbitmq.client.AMQP;

import com.rabbitmq.client.Channel;

import com.rabbitmq.client.Connection;

import com.rabbitmq.client.ConnectionFactory;

import java.util.HashMap;

import java.util.Map;

/\*\*

\* 生产者

\*

\* @author 七夜雪

\* @date 2018-12-13 20:43

\*/

public class Producer {

public static void main(String[] args) throws Exception {

// 1. 创建连接工厂, 设置属性

ConnectionFactory factory = new ConnectionFactory();

factory.setHost("192.168.72.138");

factory.setPort(5672);

factory.setVirtualHost("/");

// 2. 创建连接

Connection connection = factory.newConnection();

// 3. 使用connection创建channel

Channel channel = connection.createChannel();

// 4. 通过channel发送消息

String msg = "hello rabbitmq!";

AMQP.BasicProperties properties = new AMQP.BasicProperties();

Map<String,Object> headers = new HashMap<String, Object>();

headers.put("name", "七夜雪");

properties = properties.builder()

// 设置编码为UTF8

.contentEncoding("UTF-8")

// 设置自定义Header

.headers(headers)

// 设置消息失效时间

.expiration("5000").build();

// 设置了消息超时时间为5秒, 5秒后消息自动删除

channel.basicPublish("", "test\_ttl\_queue", properties, msg.getBytes());

// 没有设置消息存活时间,消息存活时间根据队列来决定

channel.basicPublish("", "test\_ttl\_queue", null, msg.getBytes());

// 5. 关闭连接

channel.close();

connection.close();

}

}

## 死信队列DLX

**死信队列(DLX Dead-Letter-Exchange)：**利用DLX，当消息在一个队列中变成死信(dead message)之后，它能被重新publish到另一个Exchange，这个Exchange就是DLX。

DLX也是一个正常的Exchange，和一般的Exchange没有区别，它能在任何的队列上被指定，实际上就是设置某个队列的属性。

当这个队列中有死信时，RabbitMQ就会自动的将这个消息重新发布到设置的Exchange上去，进而被路由到另一个队列。

可以监听这个队列中消息做相应的处理，这个特性可以弥补RabbitMQ3.0之前支持的immediate参数的功能。

### 消息变成死信的几种情况

* 消息被拒绝(basic.reject/basic.nack)并且requeue=false
* 消息TTL过期
* 队列达到最大长度

**死信队列设置：**需要设置死信队列的exchange和queue，然后通过routing key进行绑定。**只不过我们需要在队列加上一个参数即可**。

Map<String, Object> arguments = Maps.newHashMapWithExpectedSize(3);

arguments.put("x-message-ttl", dlx-ttl);

arguments.put("x-dead-letter-exchange","exchange-name");

arguments.put("x-dead-letter-routing-key", "routing-key");

Queue ret = QueueBuilder.durable("queue-name".withArguments(arguments).build();

只需要通过监听该死信队列即可处理死信消息。还可以通过死信队列完成延时队列。

### 死信队列的设置

1. 首先要设置死信队列的exchange和queue, 然后进行绑定
   * Exchange : dlx.exchange
   * Queue : dlx.queue
   * RoutingKey : #
2. 然后正常声明交换机, 队列, 绑定, 只不过需要在队列加上一个扩展参数即可 : arguments.put(“x-dead-letter-exchange”, “dlx.exchange”);
3. 这样消息在过期, reject或nack(requeue要设置成false), 队列在达到最大长度时, 消息就可以直接路由到死信队列

### 代码演示

1. 消费者中设置死信队列和正常队列, 启动之后关闭消费者
2. 生产者生产的消息一个超时时间, 使消息超时之后变为死信

消费者中设置死信队列

package com.qiyexue.api.dlx;

import com.rabbitmq.client.Channel;

import com.rabbitmq.client.Connection;

import com.rabbitmq.client.ConnectionFactory;

import java.util.HashMap;

import java.util.Map;

/\*\*

\* 消费者

\*

\* @author 七夜雪

\* @date 2018-12-15 20:07

\*/

public class Consumer {

public static void main(String[] args) throws Exception {

// 1. 创建连接工厂并设置属性

ConnectionFactory factory = new ConnectionFactory();;

factory.setHost("192.168.72.138");

factory.setPort(5672);

factory.setVirtualHost("/");

// 2. 创建连接

Connection connection = factory.newConnection();

// 3. 创建channel

Channel channel = connection.createChannel();

// 4. 声明死信队列Exchange和Queue

channel.exchangeDeclare("dlx.exchange", "topic");

channel.queueDeclare("dlx.queue", true, false, false, null);

channel.queueBind("dlx.queue", "dlx.exchange", "#");

// 5. 声明普通Exchange

String exchangeName = "test\_dlx\_exchange";

String exchangeType = "topic";

String routingKey = "dlx.\*";

channel.exchangeDeclare(exchangeName, exchangeType, true, false, null);

// 6. 声明消息队列, 指定死信队列为dlx.exchange

String queueName = "test\_dlx\_queue";

Map<String, Object> arguments = new HashMap<>();

// x-dead-leeter-exchange属性用于指定死信队列为dlx.exchange

arguments.put("x-dead-letter-exchange", "dlx.exchange");

channel.queueDeclare(queueName, true, false, false, arguments);

// 6. 绑定队列和Exchange

channel.queueBind(queueName, exchangeName, routingKey);

}

}

生产者消息设置超时时间

package com.qiyexue.api.dlx;

import com.rabbitmq.client.\*;

import java.util.HashMap;

import java.util.Map;

/\*\*

\* 生产者

\*

\* @author 七夜雪

\* @date 2018-12-15 19:56

\*/

public class Producer {

public static void main(String[] args) throws Exception {

// 1. 创建ConnectionFactory, 并设置属性

ConnectionFactory factory = new ConnectionFactory();

factory.setHost("192.168.72.138");

factory.setPort(5672);

factory.setVirtualHost("/");

// 2. 创建连接

Connection connection = factory.newConnection();

// 3. 创建channel

Channel channel = connection.createChannel();

String exchangeName = "test\_dlx\_exchange";

String routingKey = "dlx.qiye";

AMQP.BasicProperties properties = new AMQP.BasicProperties().builder().expiration("5000").build();

// 发送消息

String msg = "Hello, 七夜雪";

channel.basicPublish(exchangeName, routingKey, true, properties, msg.getBytes());

// 关闭连接

channel.close();

connection.close();

}

}

## 消费端ACK与NACK

消费端进行消费的时候，如果由于业务异常可以进行日志的记录，然后进行补偿。由于服务器宕机等严重问题，我们需要手动进行ACK保障消费端消费成功。

**消费端重回队列**是为了对没有成功处理消息，把消息重新返回到Broker。一般来说，实际应用中都会关闭重回队列，也就是设置为false。

// deliveryTag：消息在mq中的唯一标识

// multiple：是否批量(和qos设置类似的参数)

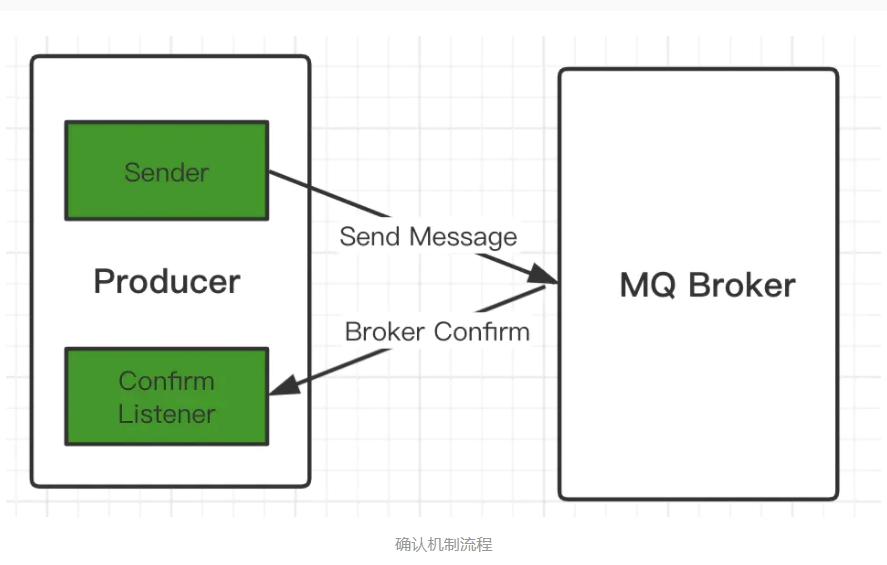
// requeue：是否需要重回队列。或者丢弃或者重回队首再次消费。

public void basicNack(long deliveryTag, boolean multiple, boolean requeue)

## 生产者Confirm机制

* 消息的确认，是指生产者投递消息后，如果Broker收到消息，则会给我们生产者一个应答。
* 生产者进行接受应答，用来确认这条消息是否正常的发送到了Broker，这种方式也是消息的可靠性投递的核心保障！

如何实现Confirm确认消息？



1、在channel上开启确认模式：channel.confirmSelect()  
2、在channel上开启监听：addConfirmListener，监听成功和失败的处理结果，根据具体的结果对消息进行重新发送或记录日志处理等后续操作。

## Return消息机制

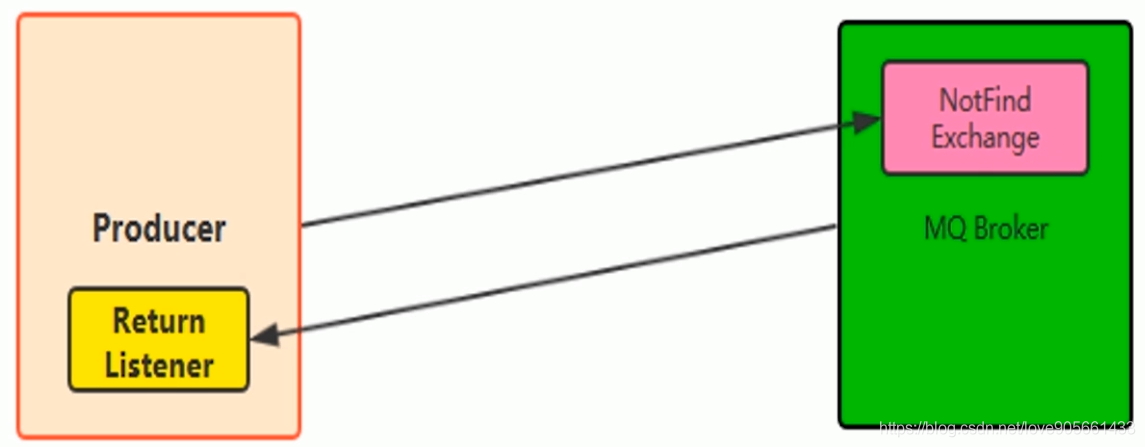
Return Listener**用于处理一些不可路由的消息**。

我们的消息生产者，通过指定一个Exchange和Routing，把消息送达到某一个队列中去，然后我们的消费者监听队列进行消息的消费处理操作。

但是在某些情况下，如果我们在发送消息的时候，当前的exchange不存在或者指定的路由key路由不到，这个时候我们需要监听这种不可达消息，就需要使用到Returrn Listener。

基础API中有个关键的配置项Mandatory：如果为true，监听器会收到路由不可达的消息，然后进行处理。如果为false，broker端会自动删除该消息。

通过chennel.addReturnListener(ReturnListener rl)传入已经重写过handleReturn方法的ReturnListener。



## 消费端自定义监听(推模式和拉模式pull/push)

 一般通过while循环进行consumer.nextDelivery()方法进行获取下一条消息进行那个消费。(通过while将拉模式模拟成推模式，但是死循环会耗费CPU资源。)

 通过自定义Consumer，实现更加方便、可读性更强、解耦性更强的方式。(现默认使用的模式，直接订阅到queue上，如果有数据，就等待mq推送过来)

Basic.Consume将信道(Channel)置为接收模式，直到取消队列的订阅为止。

在接受模式期间，RabbitMQ会不断的推送消息给消费者。

当然推送消息的个数还是受Basic.Qos的限制。

如果只想从队列获得单条消息而不是持续订阅，建议还是使用Basic.Get进行消费。

但是不能将Basic.Get放在一个循环里来代替Basic.Consume，这样会严重影响RabbitMQ的性能。

如果要实现高吞吐量，消费者理应使用Basic.Consume方法。



## 保证幂等性

消费端实现幂等性，就意味着我们的消息永远不会消费多次，即使我们收到了多条一样的信息。

* 唯一ID+指纹码机制，利用数据库主键去重
* select count(1) from table where id = id+指纹码
* 优点：实现简单
* 缺点：高并发下有数据库写入的性能瓶颈
* 解决：跟进ID进行分库分表进行算法路由
* 利用redis的原子性去实现
* 问题1：是否需要落库。如果落库，如何保证数据的一致性和原子性？
* 问题2：如果不进行落库，缓存种的数据如果设置定时同步的策略？

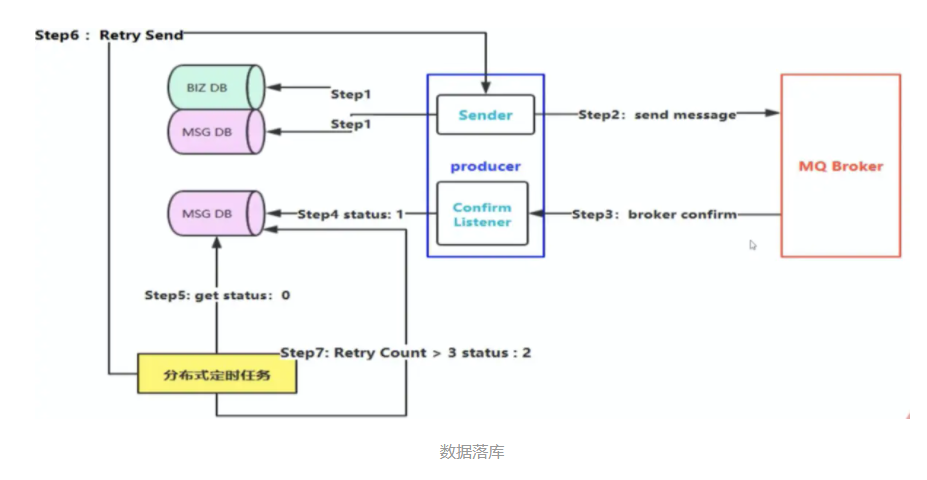
## 如何保证可靠性？

#### 生产端的可靠性投递

* 保证消息的成功发出
* 保障MQ节点的成功接受
* 发送端收到MQ节点(Broker)确认应答
* 完善消息的补偿机制

解决方案:

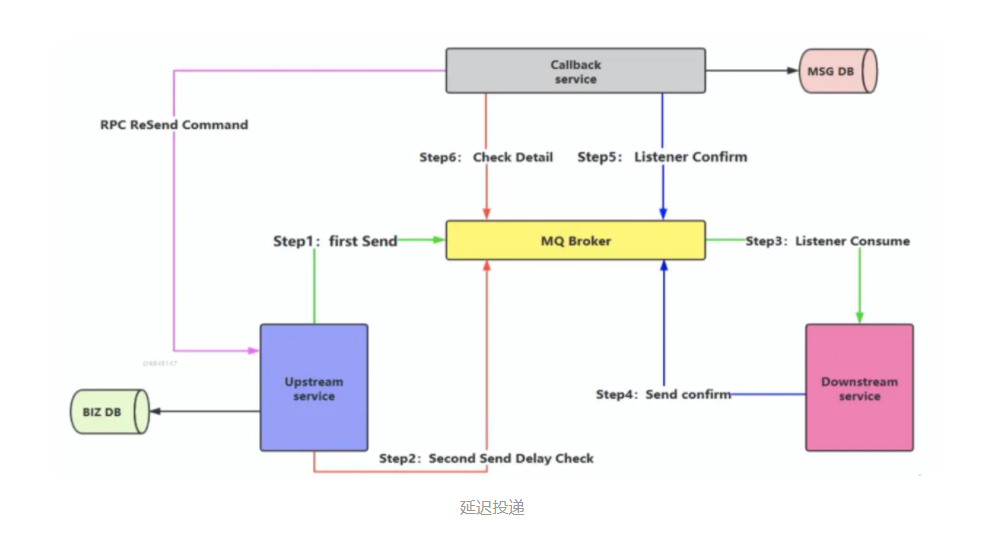
消息落库，对消息状态进行变更。



1. 消息入库
2. 消息发送
3. 消费端消息确认
4. 更新库中消息状态为已确认
5. 定时任务读取数据库中未确认的消息
6. 未收到确认结果的消息重新发送
7. 如果重试几次之后仍然失败, 则将消息状态更改为投递失败的终态, 后面需要人工介入

缺点：对数据库有多次操作。不适用于高并发业务。

消息的延迟投递，做二次确认，回调检查。



1. step1 : 第一次消息发送, 必须业务数据落库之后才能进行消息发送
2. step2 : 第二次消息延迟发送, 设定延迟一段时间发送第二次check消息
3. step3 : 消费端监听Broker, 进行消息消费
4. step4 : 消费成功之后, 发送确认消息到确认消息队列
5. step5 : Callback Service监听step4中的确认消息队列, 维护消息状态, 是否消费成功等状态
6. step6 : Callback Service监听step2发送的Delay Check的消息队列, 检测内部的消息状态, 如果消息是发送成功状态, 则流程结束, 如果消息是失败状态, 或者查不到当前消息状态时, 会通知生产者, 进行消息重发, 重新上述步骤

拆出一个回调服务。将落库、检查等操作安排至回调服务上。

1：发送者发送信息至MQ，消费者为下游业务方。

1.1：成功后，作为发送者发送信息至MQ，消费者为回调服务。

1.1.1 回调服务接受数据后，落库。

1.2：失败，等待发送者的延时投递信息。

2、发送者发送延迟投递信息至MQ，消费者为回调服务。

1.1：查库，确认下游服务方消费已成功。

1.2：查库，确认下游服务方消费失败，通过rpc调用发送者的接口重新发送。

消息发送者发送的两条信息是同时发送的。

减少了对库的操作，同时解耦，保证了性能，不能百分百保证可靠性

## 消费端如何限流

当海量消息瞬间推送过来，单个客户端无法同时处理那么多数据，严重会导致系统宕机。这时，需要削峰。

RabbitMQ提供了一种qos(服务质量保证)功能。即**在非自动确认消息的前提下**(非ACK)，如果一定数目的消息(通过基于consume或者channel设置qos的值)未被确认前，不进行消费新的消息。

* 自动签收要设置成false, 建议实际工作中也设置成false
* void basicQos(int prefetchSize, int prefetchCount, boolean global) throws IOException;
* prefetchSize : 消息大小限制, 一般设置为0, 消费端不做限制
* prefetchCount : 会告诉RabbitMQ不要同时给一个消费者推送多于N个消息, 即一旦有N个消息还没有ack, 则该consumer将block(阻塞), 直到有消息ack
* global : true/false 是否将上面设置应用于channel, 简单来说就是上面的限制是channel级别的还是consumer级别

**注意 :**

* prefetchSize和global这两项, RabbitMQ没有实现, 暂且不关注, prefetchCount在autoAck设置false的情况下生效, 即在自动确认的情况下这两个值是不生效的

// prefetchSize：消息体大小限制；0为不限制

// prefetchCount：RabbitMQ同时给一个消费者推送的消息个数。即一旦有N个消息还没有ack，则该consumer将block掉，直到有消息ack。默认是1.

// global：限流策略的应用级别。consumer[false]、channel[true]。

void BasicQos(unit prefetchSize, unshort prefetchCount, bool global);

channel.basicQos(...);

## Channel模式和Connection模式

Connection和Channel是spring-amqp中的概念，并非rabbitmq中的概念，官方文档对Connection和Channel有这样的描述：

Sharing of the connection

is possible since the "unit of work" for messaging with AMQP

is actually a "channel"

(in some ways,

this is similar to the relationship

between a Connection and a Session in JMS).

**CHANNEL模式：**程序运行期间ConnectionFactory会维护着一个Connection，所有的操作都会使用这个Connection，但一个Connection中可以有多个Channel，操作rabbitmq之前都必须先获取到一个Channel，否则就会阻塞（可以通过setChannelCheckoutTimeout()设置等待时间），这些Channel会被缓存（缓存的数量可以通过setChannelCacheSize()设置）；

**CONNECTION模式：**这个模式下允许创建多个Connection，会缓存一定数量的Connection，每个Connection中同样会缓存一些Channel，除了可以有多个Connection，其它都跟CHANNEL模式一样。

关于CONNECTION模式中，可以存在多个Connection的使用场景，官方文档的描述：

The use of separate connections

might be useful in some environments,

such as consuming from an HA cluster,

in conjunction with a load balancer,

to connect to different cluster members.

setChannelCacheSize：设置每个Connection中（注意是每个Connection）可以缓存的Channel数量，注意只是缓存的Channel数量，不是Channel的数量上限，操作rabbitmq之前（send/receive message等）要先获取到一个Channel，获取Channel时会先从缓存中找闲置的Channel，如果没有则创建新的Channel，当Channel数量大于缓存数量时，多出来没法放进缓存的会被关闭。

注意，改变这个值不会影响已经存在的Connection，只影响之后创建的Connection。  
有时会出现connection closed错误。rabbitTemplate作者对于这种问题的解决方案，他给的方案很简单，单纯的增加connection数：

connectionFactory.setChannelCacheSize(100);

setChannelCheckoutTimeout：当这个值大于0时，channelCacheSize不仅是缓存数量，同时也会变成数量上限，从缓存获取不到可用的Channel时，不会创建新的Channel，会等待这个值设置的毫秒数，到时间仍然获取不到可用的Channel会抛出AmqpTimeoutException异常。

同时，在CONNECTION模式，这个值也会影响获取Connection的等待时间，超时获取不到Connection也会抛出AmqpTimeoutException异常。

## 消费端ACK与重回队列

#### 消费端ACK

* 消费端的手工ACK和NACK, ACK是确认成功消费, NACK表示消息处理失败, 会重发消息
* 消费端进行消费的时候, 如果由于业务异常我们可以进行日志的记录, 然后进行补偿
* 如果由于服务器宕机等严重问题, 就需要手工进行ACK保障消费端消费成功

#### 重回队列

* 消费端重回队列是为了对没有处理成功的消息, 把消息重新回递给Broker
* 一般在实际应用中, 都会关闭重回队列, 也就是设置为False

## 代码演示

### 自定义消费者

package com.qiyexue.api.ack;

import com.rabbitmq.client.AMQP;

import com.rabbitmq.client.Channel;

import com.rabbitmq.client.DefaultConsumer;

import com.rabbitmq.client.Envelope;

import java.io.IOException;

import java.util.concurrent.TimeUnit;

/\*\*

\* 自定义消费者

\* 设置手动ack和重回队列

\* @author 七夜雪

\* @date 2018-12-16 8:20

\*/

public class MyConsumer extends DefaultConsumer {

private Channel channel;

public MyConsumer(Channel channel) {

super(channel);

this.channel = channel;

}

@Override

public void handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, byte[] body) throws IOException {

System.out.println("-------------自定义消费者------------");

System.out.println("consumerTag : " + consumerTag);

System.out.println("envelope : " + envelope);

System.out.println("properties : " + properties);

System.out.println("body : " + new String(body));

try {

TimeUnit.SECONDS.sleep(2);

} catch (InterruptedException e) {

e.printStackTrace();

}

if ((Integer)properties.getHeaders().get("num") == 0) {

// 三个参数 : DeliveryTag, 是否批量拒绝, 是否可以重回队列

channel.basicNack(envelope.getDeliveryTag(), false, true);

} else {

channel.basicAck(envelope.getDeliveryTag(), false);

}

}

}

### Producer

package com.qiyexue.api.ack;

import com.rabbitmq.client.\*;

import java.util.HashMap;

import java.util.Map;

/\*\*

\* 生产者

\*

\* @author 七夜雪

\* @date 2018-12-15 19:56

\*/

public class Producer {

public static void main(String[] args) throws Exception {

// 1. 创建ConnectionFactory, 并设置属性

ConnectionFactory factory = new ConnectionFactory();

factory.setHost("192.168.72.138");

factory.setPort(5672);

factory.setVirtualHost("/");

// 2. 创建连接

Connection connection = factory.newConnection();

// 3. 创建channel

Channel channel = connection.createChannel();

String exchangeName = "test\_ack\_exchange";

String routingKey = "ack.qiye";

for (int i = 0; i < 5; i++) {

// 发送消息

String msg = "Hello, 七夜雪 " + i;

Map<String, Object> hearder = new HashMap<String, Object>();

hearder.put("num", i);

AMQP.BasicProperties properties = new AMQP.BasicProperties().builder()

.contentEncoding("UTF-8")

.headers(hearder).build();

channel.basicPublish(exchangeName, routingKey, true, properties, msg.getBytes());

}

// 关闭连接

channel.close();

connection.close();

}

}

### Consumer

package com.qiyexue.api.ack;

import com.rabbitmq.client.Channel;

import com.rabbitmq.client.Connection;

import com.rabbitmq.client.ConnectionFactory;

/\*\*

\* 消费者，关闭自动ack

\*

\* @author 七夜雪

\* @date 2018-12-15 20:07

\*/

public class Consumer {

public static void main(String[] args) throws Exception {

// 1. 创建连接工厂并设置属性

ConnectionFactory factory = new ConnectionFactory();;

factory.setHost("192.168.72.138");

factory.setPort(5672);

factory.setVirtualHost("/");

// 2. 创建连接

Connection connection = factory.newConnection();

// 3. 创建channel

Channel channel = connection.createChannel();

// 4. 声明Exchange

String exchangeName = "test\_ack\_exchange";

String exchangeType = "topic";

String routingKey = "ack.\*";

channel.exchangeDeclare(exchangeName, exchangeType, true, false, null);

// 5. 声明消息队列

String queueName = "test\_ack\_queue";

channel.queueDeclare(queueName, true, false, false, null);

// 6. 绑定队列和Exchange

channel.queueBind(queueName, exchangeName, routingKey);

// 7. 设置消费者为自定义的消费者, 将autoAck设置为false

channel.basicConsume(queueName, false, new MyConsumer(channel));

}

}

## RabbitMQ集群

RabbitMQ最优秀的功能之一就是内建集群，这个功能涉及的目的是允许消费者和生产者在节点崩溃的情况下继续运行，以及通过添加更多的节点来线性扩展消息通信吞吐量。RabbitMQ内部利用Erlang提供的分布式通信框架OTP来满足上述需求，使客户端在失去一个RabbitMQ节点连接的情况下，还是能够重新连接到集群中的其他节点继续胜场、消费信息。

RabbitMQ会始终记录以下四中类型的内部元数据：

* 队列元数据：包括队列名称和他们的属性，比如是否可持久化，是否可持久化，是否自动删除。
* 交换器元数据：交换器名称、类型、属性。
* 绑定元数据：内部是一张表格，记录如何将消息路由到队列。
* vhost元数据：为vhost内部的队列、交换器、绑定提供命名空间和安全属性。

在单一节点中，RabbitMQ会将所有这些信息存储在内存中，同时将标记为可持久化的队列、交换器、 绑定存储在硬盘上。存到硬盘上可以确保队列和交换器在节点重启后能够重建。**而在集群模式下，同样也提供了两种选择：存到硬盘上(独立节点的默认配置)，存在内存中。**

如果在集群中创建队列，集群只会在单个节点而不是所有节点上创建完整的队列信息（元数据、状态、内容）。结果是只有队列的所有者节点知道有关队列的所有信息，因此当集群节点崩溃时，该节点的队列和绑定就消失了，并且任何匹配该队列的绑定的新消息也丢失了。还好RabbitMQ 2.6.0之后提供了镜像队列以避免集群节点故障导致的队列内容不可用。

RabbitMQ 集群中可以共享 user、vhost、exchange等，所有的数据和状态都是必须在所有节点上复制的，例外就是上面所说的消息队列。RabbitMQ 节点可以动态的加入到集群中。

当在集群中声明队列、交换器、绑定的时候，这些操作会直到所有集群节点都成功提交元数据变更后才返回。集群中有内存节点和磁盘节点两种类型，内存节点虽然不写入磁盘，但是它的执行比磁盘节点要好。内存节点可以提供出色的性能，磁盘节点能保障配置信息在节点重启后仍然可用，那集群中如何平衡这两者呢？

RabbitMQ 只要求集群中至少有一个磁盘节点，所有其他节点可以是内存节点，当节点加入或离开集群时，它们必须要将该变更通知到至少一个磁盘节点。如果只有一个磁盘节点，刚好又是该节点崩溃了，那么集群可以继续路由消息，但不能创建队列、创建交换器、创建绑定、添加用户、更改权限、添加或删除集群节点。换句话说集群中的唯一磁盘节点崩溃的话，集群仍然可以运行，但直到该节点恢复，否则无法更改任何东西。

## RabbitMQ集群恢复与故障转移

前提 : A, B两个节点组成一个镜像队列, B是Master节点

场景一

* A先停, B后停
* 解决方案 : 该场景下B是Master, 只要先启动B, 在启动A即可。或者先启动A, 30秒之内启动B即可恢复镜像队列

**场景二**

* A, B同时停机
* 解决方案 : 只需要在30秒内连续启动A和B即可恢复镜像

**场景三**

* A先停, B后停, 且A无法恢复
* 解决场景 : 因为B是Master, 所以等B启起来以后, 在B节点上调用控制台命令 : **rabbitmqctl forget\_cluster\_node A**解除与A的Cluster关系, 再将新的Slave节点加入B即可重新恢复镜像队列

**场景四**

* A先停, B后停, 且B无法恢复
* 解决方案 : 因为Master节点无法恢复, 所以较难处理, 在3.4.2之前没有什么好的解决方案, 但是现在已经有解决方案了, 在3.4.2以后的版本。因为B是主节点, 所以直接启动A是不行的, 当A无法启动时, 也就没有办法在A节点上调用**rabbitmqctl forget\_cluster\_node B 命令了。但是在新版本中forget\_cluster\_node支持–offline**参数, 支持线下移除节点。这就意味着运行rabbitmqctl在理想节点上执行命令, 迫使RabbitMQ在未启动Slave节点中选择一个节点作为Master。当在A节点执行\*\*rabbitmqctl forget\_cluster\_node --offline B \*\*时, RabbitMQ会mock一个节点代表A, 执行 **forget\_cluster\_node**命令将B移除cluster, 然后A就可以正常启动了, 最后将新的Slave节点加入A即可重新恢复镜像队列

**场景五**

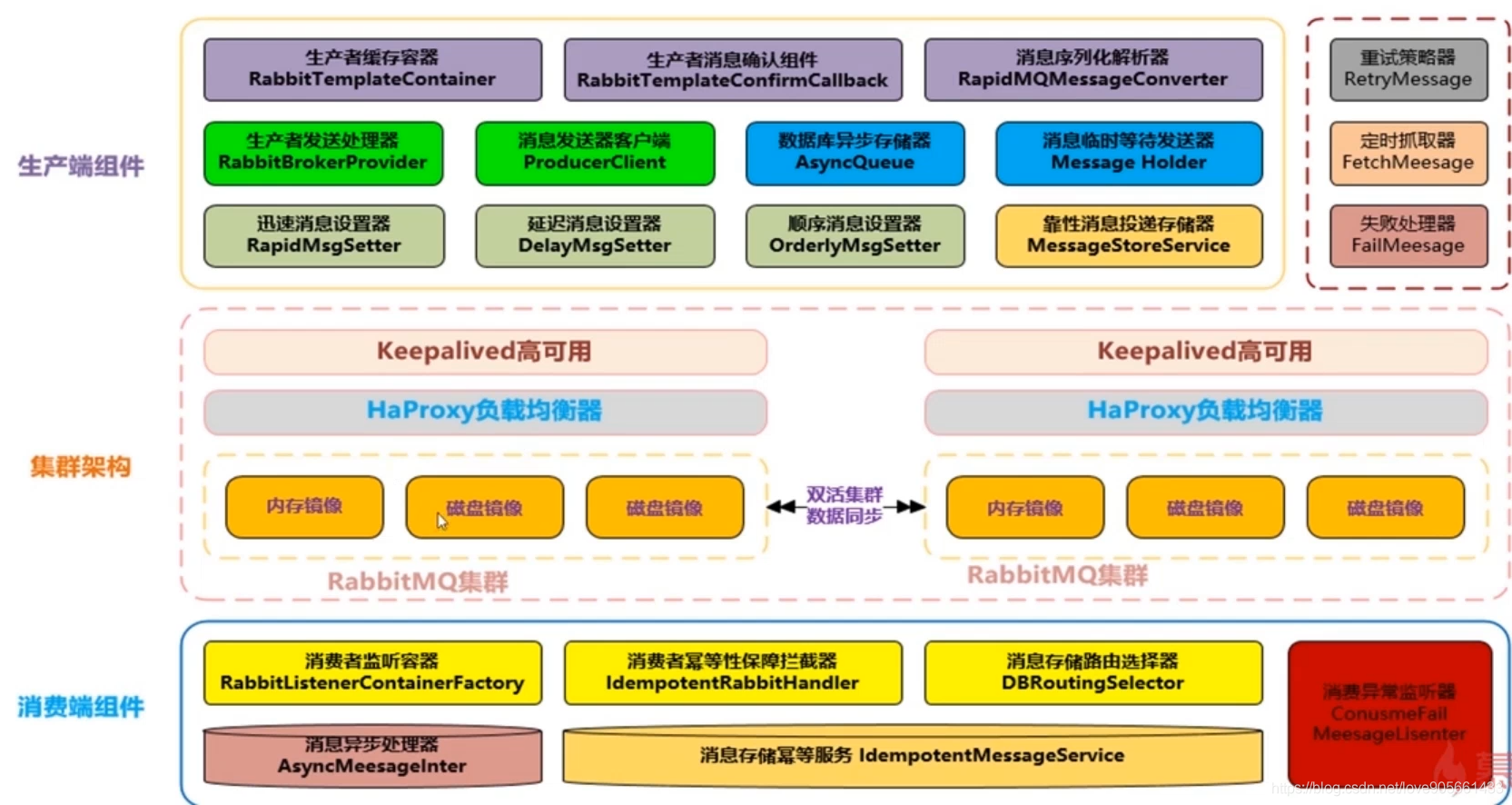
* A先停, B后停, 且A, B均无法恢复, 但是能得到A或B的磁盘文件
* 解决方案 : 这种场景更加难以处理, 只能通过恢复数据的方式去尝试恢复, 将A或B的数据库文件默认在$RABBIT\_HOME/var/lib目录中, 把它拷贝到新节点对应的目录下, 再将新节点的hostname改成A或B的hostname, 如果是A节点(Slave)的磁盘文件, 按照场景四处理即可, 如果是B节点(Master)的磁盘文件, 则按照场景三处理, 最后将新的Slave加入到新节点后完成恢复
* 这种场景很极端, 只能尝试恢复

**场景六**

* A,B均停机, A,B均无法恢复, 且A或B的磁盘文件都无法恢复, 极端情况
* 解决方案 : 一首凉凉送给大家, 这种情况完全没办法解决

## 基础组件封装

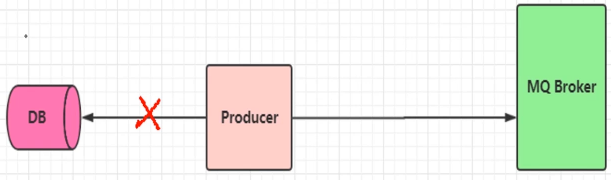
### 基础组件封装思路和架构设计方案



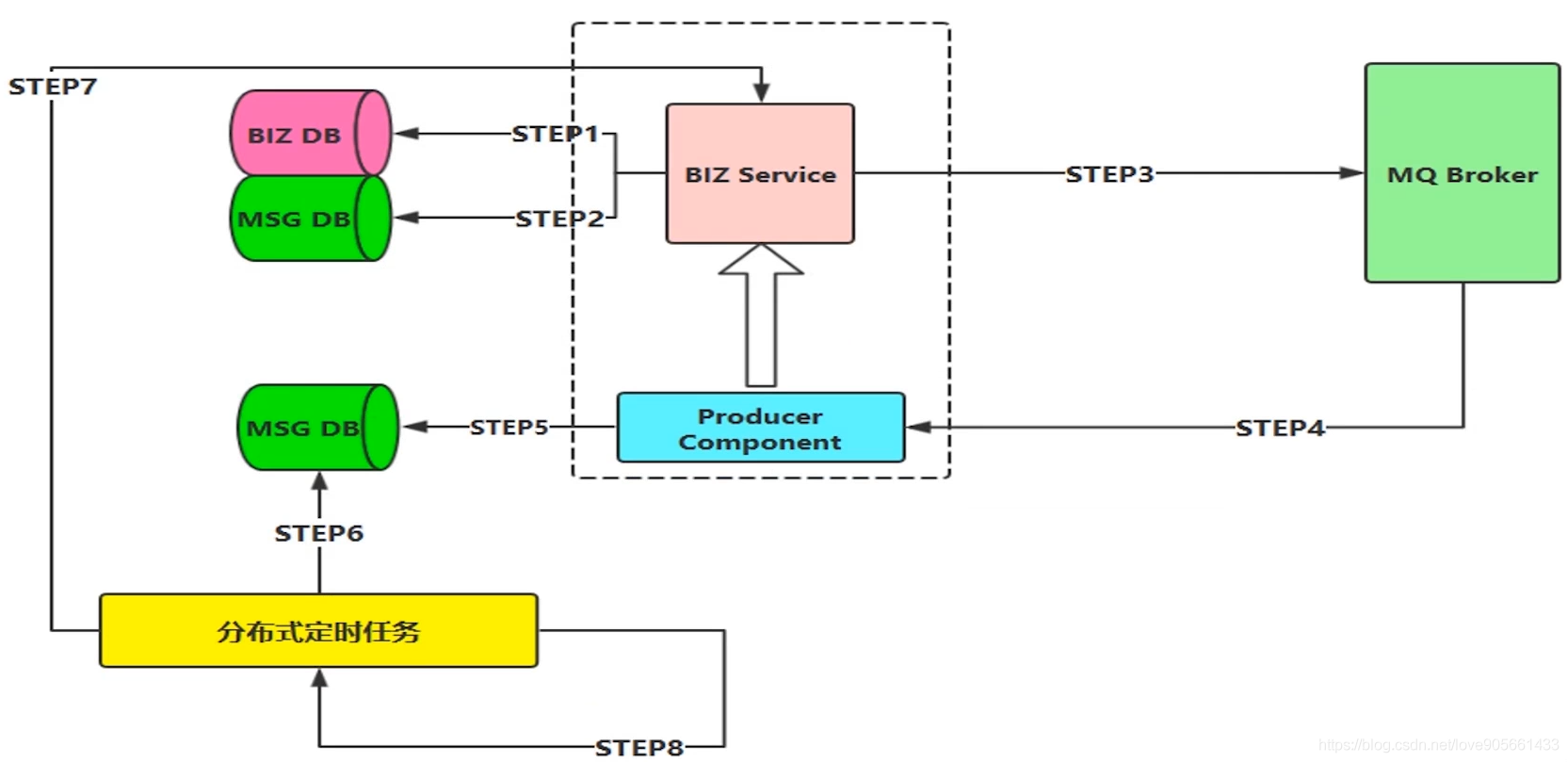
* 支持高性能的序列化转换, 异步化发送消息
* 支持消息生产实例与消费实例的链接池化缓存化, 提升性能
* 支持可靠性投递消息, 保障消息100%不丢失
* 支持消费端的幂等操作, 避免消费端重复消费的问题
* 支持迅速消息发送模式, 在一些日志收集/统计分析等需求下可以保证高性能, 高吞吐量
* 支持延迟消息模式, 消息可以延迟发送, 指定延迟时间, 用于某些延迟检查, 服务限流场景
* 支持事务消息, 器100%保障可靠性投递, 在金融行业单笔大金额操作时会有此类需求
* 支持顺序消息, 保证消息送达消费端的先后顺序
* 支持消息补偿, 重试, 以及快速定位异常/失败消息
* 支持集群消息负载均衡, 保障消息落到具体SET集群的负载均衡
* 支持消息路由策略, 指定某些消息路由到指定的SET集群

### 迅速消息发送

* 迅速消息是指消息不进行落库存储, 不做可靠性的保障
* 在一些非核心消息, 日志数据, 或者统计分析等场景下比较合适
* 迅速消息的优点就是性能最高, 吞吐量最大



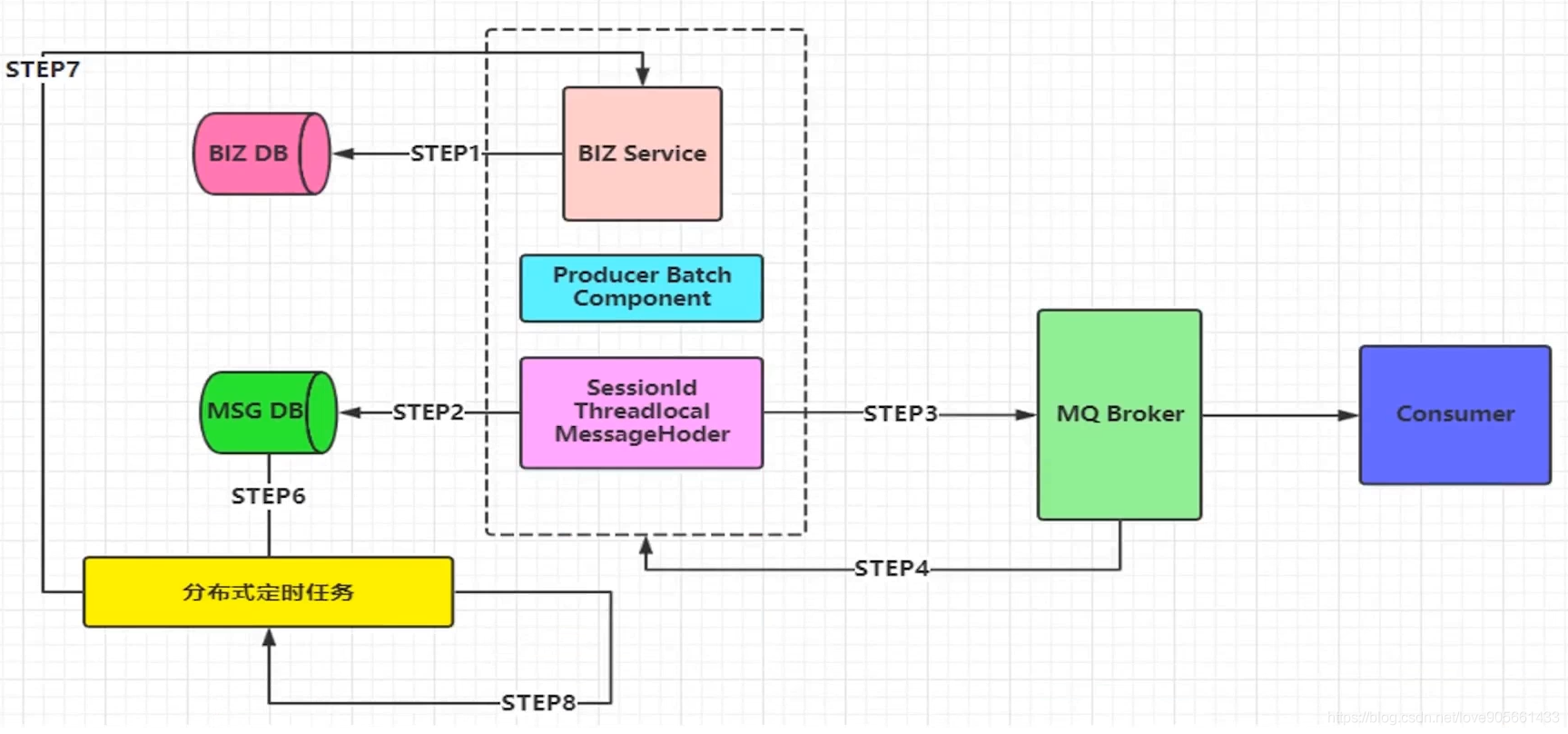
### 确认消息发送



1. step1, step2 : 业务信息和消息信息入库
2. step3 : 消息发送到MQ Broker
3. step4 : Broker回送一个ACK确认消息
4. step5 : 更新DB中消息状态
5. step6 : 定时任务读取中间状态消息
6. step7 : 执行消息重发
7. step8 : 重发多次依旧失败的消息, 将其置为失败终态

### 批量消息发送

批量消息是指我们把消息放到一个集合里统一进行提交, 这种方案设计思路是期望消息在一个会话里, 比如投递到threadlocal里的集合, 然后拥有相同的会话ID, 并且带有这次提交消息的SIZE等相关相关属性, 最重要的一点是要把这一批消息进行合并。对于Channel而言, 就是发送一次消息。这种方式也是希望消费端在消费的时候, 可以进行批量化的消费, 针对于某一个原子业务的操作去处理, 但是不保障可靠性, 需要进行补偿机制。

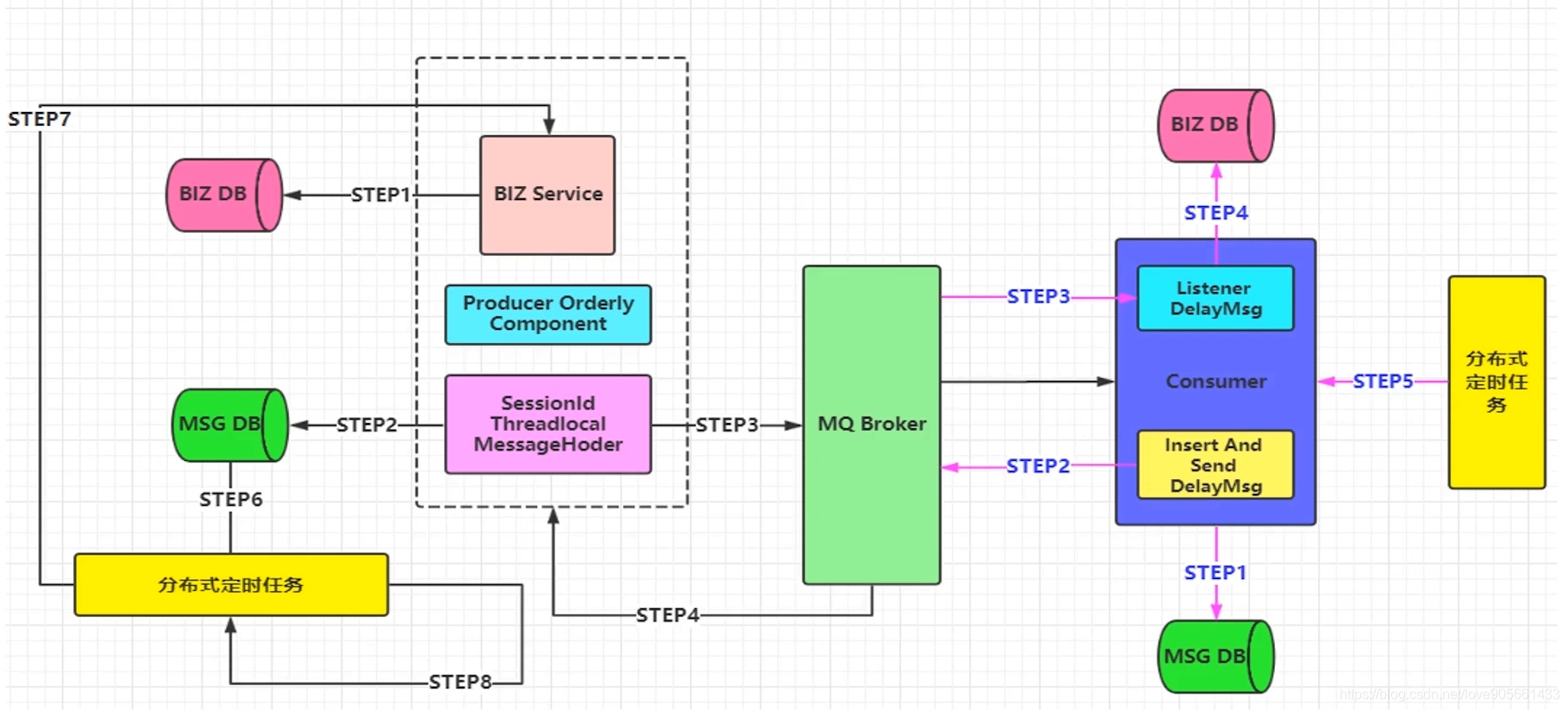


1. step1 : 业务数据入库
2. step2 : 消息组装之后进行统一入库(如果不需要可靠性投递的话, 可以省略)
3. step3 : 消息发送到Broker
4. step4-step8基本和确认消息是一致的,如果不需要可靠性投递的话, 是可以省略的

### 延迟消息发送

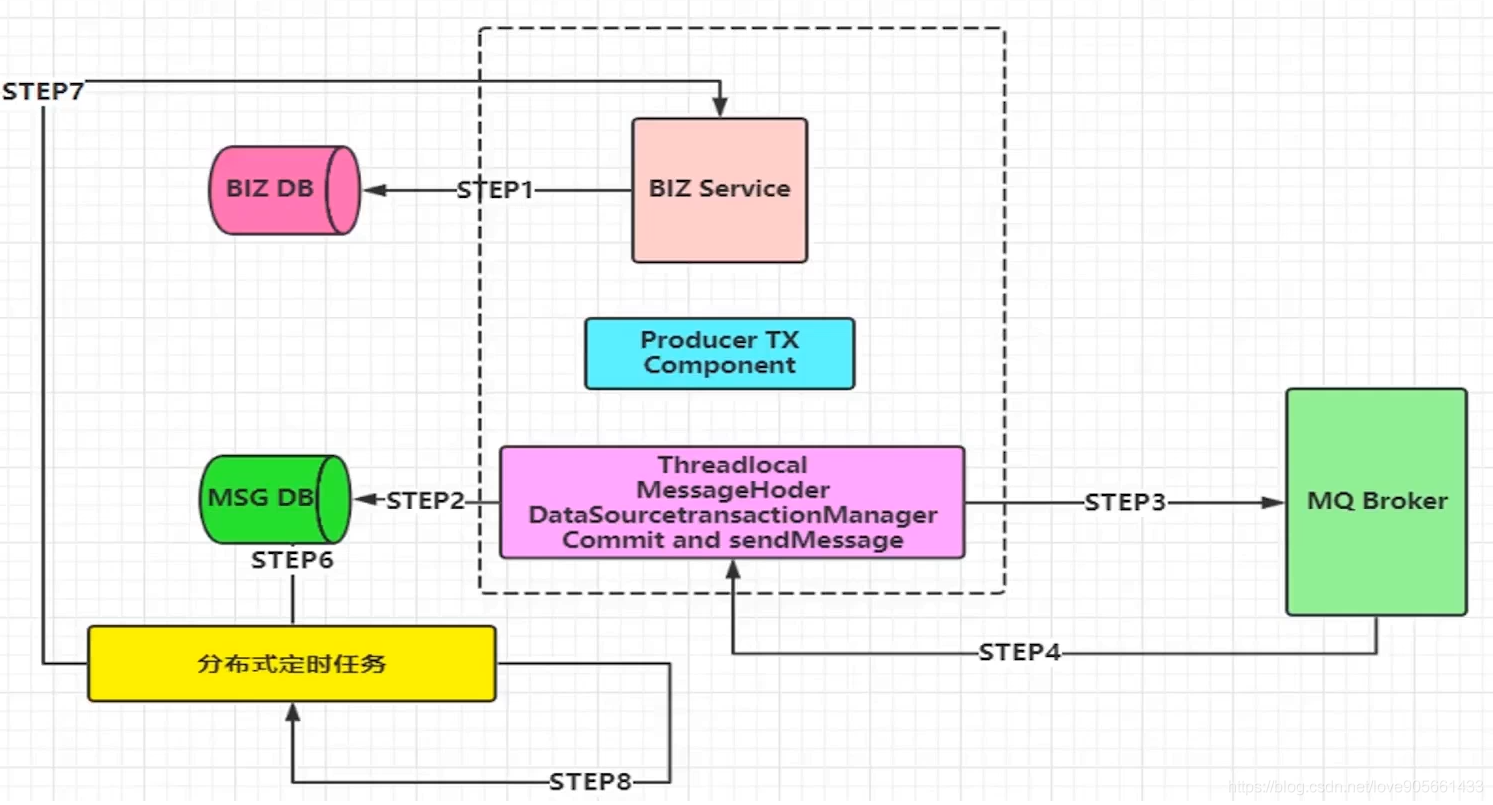
* 延迟消息就是在Message封装的时候, 添加delayTime属性即可, 使得我们的消息可以进行延迟发送1

### 顺序消息发送

* 顺序消息有点类似于批量消息的实现机制, 但是有些不同, 顺序消息需要保障一下几点:
  1. 发送的顺序消息, 必须保障消息投递到同一个队列, 且这个消费者只能有一个(独占模式)
  2. 然后需要统一提交(可能是合并成一个大的消息, 也可能是拆分为多个消息), 并且所有消息的会话ID要一致
  3. 添加消息属性 : 顺序标记的序号, 和本次顺序消息的SIZE属性, 进行落库操作
  4. 并行进行发送给自身的延迟消息(带上关键属性 : **会话ID, SIZE**)进行后续处理消费
  5. 当收到延迟消息后, 根据会话**ID, SIZE**抽取数据库数据进行处理即可
  6. 定时轮询补偿机制, 对于异常情况(如生产端消息没有完全投递成功或者消费端落库异常导致消费端落库后缺少消息条目的情况)进行补偿  
     

### 事务消息发送

* 事务消息, 使用相对较少
* 为了保障性能的同时, 也支持事务。并不推荐选择传统的RabbitMQ事务和Spring集成的机制, 因为在性能测试过程中, 这种方式性能并不理想, 非常消耗系统资源, 且会出现阻塞等情况, 高峰期也是一定程度上影响MQ集群的性能
* 解决方案 : 采用类似可靠性投递的机制, 也就是补偿机制。但是数据源必须是同一个, 也就是业务操作的数据库DB1和消息记录的数据库DB2使用同一个数据库, 保证DB层的数据一致性
* 然后利用重写Spring DataSourceTransactionManager, 在本地事务提交的时候进行发送消息, 但是也有可能事务提交成功但是消息发送失败, 这个时候就需要进行补偿了。

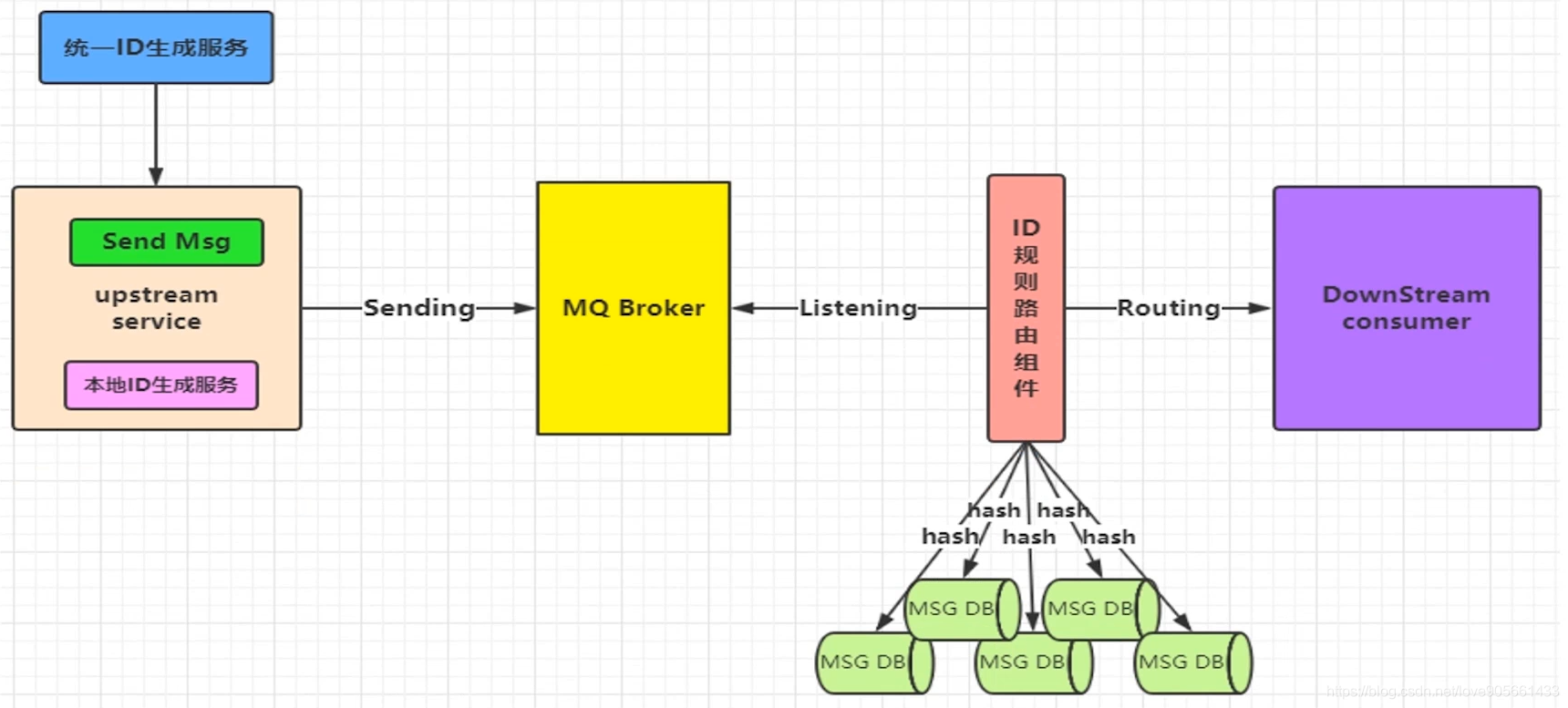


### 消息幂等性保障-消息路由规则架构设计

**可能导致消息出现非幂等性的原因 :**

* 可靠性消息投递机制造成的消息重发
* MQ Broker服务于消费端消息传输的过程中出现网络抖动
* 消费端故障或异常

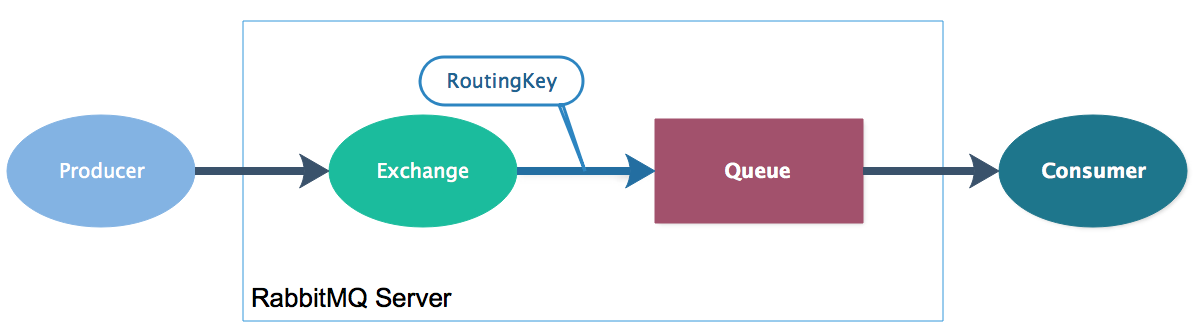
**消息幂等性设计 :**



# RabbitMQ使用分析和高可用集群搭建

## RabbitMQ 基础理解

### RabbitMQ，是一个使用 erlang 编写的 AMQP（高级消息队列协议）的服务实现，简单来说，就是一个功能强大的消息队列服务。



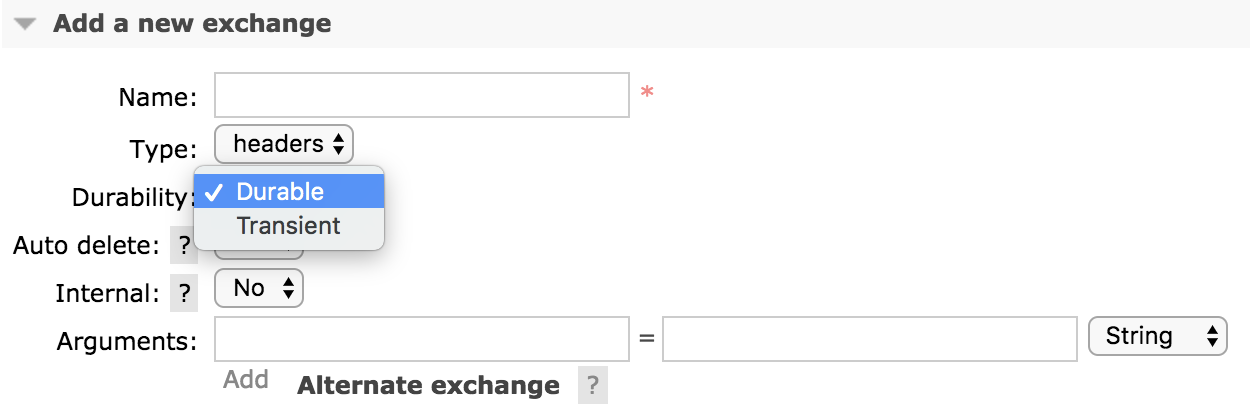
### 概念理解

* **Producer**: 消息发送者
* **RabbitMQ**:
  + Vhost: 相当于分组，每个vhost下数据是隔离的
  + Exchange: 路由器，接收消息，本根据RoutingKey分发消息
    - headers：消息头类型 路由器，内部应用
    - direct：精准匹配类型 路由器
    - topic：主题匹配类型 路由器，支持正则 模糊匹配
    - fanout：广播类型 路由器，RoutingKey无效
  + RoutingKey: 路由规则
  + Queue: 队列，用于存储消息（消息的目的地）
* **Consumer**: 消息消费者

### 持久化

    一个好的消息队列当然需要消息持久化功能，服务宕机，未消费消息不丢失，RabbitMQ持久化分为Exchange、Queue、Message  
    Exchange 和 Queue 持久化 指持久化Exchange、Queue 元数据，持久化的是自身，服务宕机，Exchange 和 Queue 自身就没有了  
    Message 持久化 顾名思义 把每一条消息体持久化，服务宕机，消息不丢失

    Durable 持久、Transient 临时，Queue新建类似



### 分析理解

    便于更直观的理解，把 RabbitMQ 的消息流对比与Http Rest接口更家熟悉形象

        www.xxx.com/webappPath/trade/getOrder -> getOrder(message) GET

RabbitMQ Server：同比 域名 www.xxx.com，只有通过域名才能到达 Server  
Vhost：同比 /webappPath，一个域名可能指向多个app  
Exchange：同比 /trade，trade/\* 下有多个method，但是需要先到达这个Class  
RoutingKey：同比 /getOrder，只有完成的 URL 才是有效的，才能确定到具体的方法 Queue：同比 getOrder(message) 消息的最终目的地  
Exchange Type:  同比 GET，但是Rest MethodType是整个URL的Type，而不是 Queue

    以上只是为了更好理解，千万不要混淆

    Producer / Consumer 就很好理解了，基于AMQP协议链接RabbitMQ Server，发送消息 / 接收消息

## RabbitMQ 消息确认策略分析

### Confrim / Transaction 概念应用

  RabbitMQ 提供了两种可靠性的确认策略 Confrim / Transaction，Producer Client仅分析Spring-Amqp，两种机制主要影响发送:  
    **Confrim**: 简单说就是直接传送消息 client > mq, 接收到 mq > ack, mq 在异步的完成 接下来的事情  
    **Transaction**: client 请求开启事务  > 发送message > client 提交事务，整个过程是同步的，mq必须完成消息持久化、消息同步等。

spring-amqp  提供的发送客户端 默认是Confrim 异步Ack模式，不用特殊配置，Transaction 需要在默认的基础上增加 RabbitMQ事务管理器

// 1.向Spring中注册RabbitMQ事务管理器

@Bean

public RabbitTransactionManager rabbitTransactionManager(ConnectionFactory connectionFactory) {

return new RabbitTransactionManager(connectionFactory);

}

...

// 2.设置通道为Transaction类型

rabbitTemplate.setChannelTransacted(true);

...

// 3.对应的方法添加@Transactional

@Transactional

public void send(String exchange, String routingKey, Object object) {

rabbitTemplate.convertAndSend(exchange, routingKey, object);

}

// 这是只是举例，具体写法和其他说明，请具体看代码注释有更详细的说明和写法component.RabbitSender



### Consumer

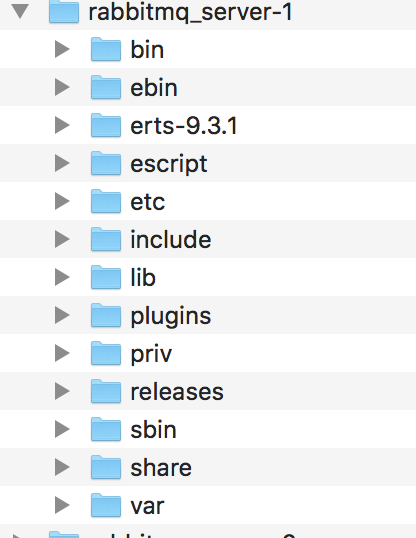
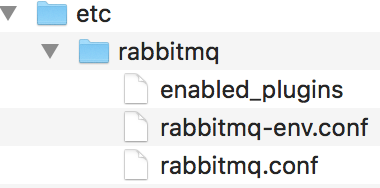
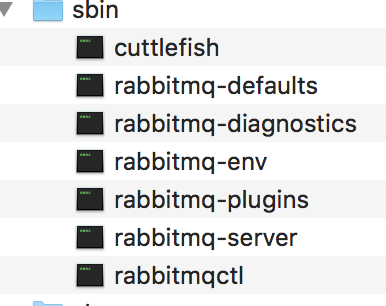
消费的机制和发送差不多, 但流程变为 Consumer 处理消息，需要Ack MQ Server, Server 才会真正的删除消息，通常消费者不需要开启事务，当处理异常抛出，Ack无法发到Server到，消息就会回到队列中，继续重试，阻塞到直到消息被消费Ack掉，所说的消息阻塞  
    具体写法和其他说明，请具体看代码注释有更详细的说明和写法component.Receiver

## RabbitMQ 配置

### RabbitMQ 安装

  RabbitMQ是基于Erlang运行的，首先选择RabbitMQ版本，确定需要的Erlang版本，然后安装Erlang，自行百度、谷歌、[RabbitMQ官网](https://www.rabbitmq.com/download.html)或者[Erlang官网](https://www.erlang-solutions.com/resources/download.html)都会有相应的资源、教程（ps: Erlang 版本请严格按照所选RabbitMQ版本要求的Erlang范围安装，否则会有各种不治之症）  
    本文以 Erlang20.03，RabbitMQ 3.7为例，RabbitMQ为linux 通用包，不同安装方式版本配置文件路径有差异，通用包好处，可移植性、控制性好

包目录结构：

    ./sbin/    rabbitmq 启动rabbitmq-server、插件rabbitmq-plugins、功能rabbitmqctl等脚本位置  
    ./etc/rabbitmq/    rabbitmq 启动配置，包括随启动插件配置、环境配置、应用配置

### RabbitMQ 配置文件

* rabbitmq-env.conf 环境配置 key = val 形式

# 指定节点的名字 默认 rabbit@${hostname}，如指定了节点名，需配置 host ip cluster1

NODENAME=rabbit@cluster1

# 指定端口 默认 5672

NODE\_PORT=5672

# 配置持久目录

MNESIA\_BASE=/mnt/data1/rabbitmq/store

# 配置日志目录 默认文件名字：${NODENAME}.log 可以用配置修改

LOG\_BASE=/mnt/data1/rabbitmq/logs

* rabbitmq.conf 环境配置 key = val 形式

        主要配置日志、默认用户信息、持久化相关等，没有定制化通常不用修改

# console log config 主要测试排查问题

log.console = false

log.console.level = debug

# log config

log.file = rabbit.log

log.file.level = info

log.file.rotation.date = $D0

# web port

management.listener.port = 15672

* enabled\_plugins 配置相应插件的名字 server start plugin也会启动

[rabbitmq\_management].

具体请看官方配置说明, 详细的讲解了rabbitmq-env.conf 和 rabbitmq.conf 配置 [官方配置说明](https://www.rabbitmq.com/configure.html)

RabbitMQ 常用命令：

# 后台启动本地服务

./rabbitmq-server –detached

# 开启/关闭 服务

./rabbitmqctl start\_app {-n node\_name}

./rabbitmqctl stop\_app {-n node\_name}

# 开启/关闭某个插件 (重启服务器后生效)

./rabbitmq-plugins enable xxx

./rabbitmq-plugins disable xxx

# 更改节点类型

./rabbitmqctl change\_cluster\_node\_type {disc/ram} {-n node\_name}

# 配置用户

./rabbitmqctl add\_user username password

./rabbitmqctl change\_password username newpassword

./rabbitmqctl delete\_user username

./rabbitmqctl set\_user\_tags username administrator

Tag: none、management、policymaker、monitoring、administrator

./rabbitmqctl set\_permissions -p /vhost1 username 'conf' 'write' 'read'

conf 一个正则表达式match哪些配置资源能够被该用户配置

write 一个正则表达式match哪些配置资源能够被该用户写入

read 一个正则表达式match哪些配置资源能够被该用户读取

## 交换机属性

* Name : 交换机名称
* Type : 交换机类型, direct, topic, fanout, headers
* Durability : 是否需要持久化, true为持久化
* Auto Delete : 当最后一个绑定到Exchange上的队列删除后, 自动删除该Exchange
* Internal : 当前Exchange是否用于RabbitMQ内部使用, 默认为False, 这个属性很少会用到
* Arguments : 扩展参数, 用于扩展AMQP协议制定化使用

## 高可用的集群搭建

### 基础概念

    RabbitMQ 集群分为两种 普通集群 和 镜像集群，可以说 镜像集群 是 普通集群 的晋升版

#### 普通集群

    以两个节点（rabbit01、rabbit02）为例来进行说明。  
    rabbit01和rabbit02两个节点仅有相同的元数据，即队列的结构，但消息实体只存在于其中一个节点rabbit01（或者rabbit02）中。  
    当消息进入rabbit01节点的Queue后，consumer从rabbit02节点消费时，RabbitMQ会临时在rabbit01、rabbit02间进行消息传输，把A中的消息实体取出并经过B发送给consumer。所以consumer应尽量连接每一个节点，从中取消息。即对于同一个逻辑队列，要在多个节点建立物理Queue。否则无论consumer连rabbit01或rabbit02，出口总在rabbit01，会产生瓶颈。当rabbit01节点故障后，rabbit02节点无法取到rabbit01节点中还未消费的消息实体。如果做了消息持久化，那么得等rabbit01节点恢复，然后才可被消费；如果没有持久化的话，就会产生消息丢失的现象。

#### 镜像集群

    在普通集群的基础上，把需要的队列做成镜像队列，消息实体会主动在镜像节点间同步，而不是在客户端取数据时临时拉取，也就是说多少节点消息就会备份多少份。该模式带来的副作用也很明显，除了降低系统性能外，如果镜像队列数量过多，加之大量的消息进入，集群内部的网络带宽将会被这种同步通讯大大消耗掉。所以在对可靠性要求较高的场合中适用  
    由于镜像队列之间消息自动同步，且内部有选举master机制，即使master节点宕机也不会影响整个集群的使用，达到去中心化的目的，从而有效的防止消息丢失及服务不可用等问题

镜像队列实现了 RabbitMQ 的高可用性（HA），具体的实现策略如下所示：

| **ha-mode** | **ha-params** | **功能** |
| --- | --- | --- |
| all | 空 | 镜像队列将会在整个集群中复制。当一个新的节点加入后，也会在这 个节点上复制一份。 |
| exactly | count | 镜像队列将会在集群上复制 count 份。如果集群数量少于 count 时候，队列会复制到所有节点上。  如果大于 Count 集群，有一个节点 crash 后，新进入节点也不会做新的镜像。 |
| nodes | node name | 镜像队列会在 node name 中复制。如果这个名称不是集群中的一个，这不会触发错误。  如果在这个 node list 中没有一个节点在线，那么这个 queue 会被声明在 client 连接的节点。 |

### 设计集群的目的

* 允许消费者和生产者在 RabbitMQ 节点崩溃的情况下继续运行。
* 通过增加更多的节点来扩展消息通信的吞吐量。

### 集群配置方式

* cluster：不支持跨网段，用于同一个网段内的局域网；可以随意的动态增加或者减少；节点之间需要运行相同版本的 RabbitMQ 和 Erlang。
* federation：应用于广域网，允许单台服务器上的交换机或队列接收发布到另一台服务器上交换机或队列的消息，可以是单独机器或集群。federation 队列类似于单向点对点连接，消息会在联盟队列之间转发任意次，直到被消费者接受。通常使用 federation 来连接 internet 上的中间服务器，用作订阅分发消息或工作队列。
* shovel：连接方式与 federation 的连接方式类似，但它工作在更低层次。可以应用于广域网。

### 节点类型

* RAM node：内存节点将所有的队列、交换机、绑定、用户、权限和 vhost 的元数据定义存储在内存中，好处是可以使得像交换机和队列声明等操作更加的快速。
* Disk node：将元数据存储在磁盘中，单节点系统只允许磁盘类型的节点，防止重启 RabbitMQ 的时候，丢失系统的配置信息。

问题说明：RabbitMQ 要求在集群中至少有一个磁盘节点，所有其他节点可以是内存节点，当节点加入或者离开集群时，必须要将该变更通知到至少一个磁盘节点。如果集群中唯一的一个磁盘节点崩溃的话，集群仍然可以保持运行，但是无法进行其他操作（增删改查），直到节点恢复。  
解决方案：设置两个磁盘节点，至少有一个是可用的，可以保存元数据的更改。

### Erlang Cookie

Erlang Cookie 是保证不同节点可以相互通信的密钥，要保证集群中的不同节点相互通信必须共享相同的 Erlang Cookie。具体的目录存放在/var/lib/rabbitmq/.erlang.cookie。

说明：这就要从 rabbitmqctl 命令的工作原理说起，RabbitMQ 底层是通过 Erlang 架构来实现的，所以 rabbitmqctl 会启动 Erlang 节点，并基于 Erlang 节点来使用 Erlang 系统连接 RabbitMQ 节点，在连接过程中需要正确的 Erlang Cookie 和节点名称，Erlang 节点通过交换 Erlang Cookie 以获得认证。

### 集群搭建

    RabbitMQ 集群通信的验证机制是通过 erlang.cookie进行确认的，只有erlang.cookie一致的两个服务才能通信，创建cookie文件：

mkdir ~/.erlang.cookie

echo 'SJJARLYPVRPMWFVGKWZZ' > ~/.erlang.cookie

chmod 400 ~/.erlang.cookie

cluster1=10.0.0.1, cluster2=10.0.0.2  2台服务器为例，本地搭建需修改 tcp端口、web端口

写入hostes：

10.0.0.1 cluster1

10.0.0.2 cluster2

修改rabbitmq-env.conf：

# server1

NODENAME=rabbit@cluster1

...

# server2

NODENAME=rabbit@cluster2

 启动server1：

./rabbitmq-server –detached

    将server2加入到server形成集群：

/rabbitmqctl -n rabbit@cluster2 stop\_app

# 重置元数据、集群配置等信息

./rabbitmqctl -n rabbit@cluster2 reset

# cluster2 加入到 cluster1 的集群中 --ram表示cluster2为RAM节点 默认为disc

./rabbitmqctl -n rabbit@cluster2 join\_cluster rabbit@cluster1 --ram

./rabbitmqctl -n rabbit@cluster2 start\_app

    普通集群就搭建完成了，普通集群并不是高可用的，基于普通集群升级为镜像集群[RabbitMQ HA方案](https://www.rabbitmq.com/ha.html)

./rabbitmqctl set\_policy <name> [-p <vhost>] <pattern> <definition> [--apply-to <apply-to>]

    name: 策略名称

    vhost: 指定vhost, 默认值 /

    pattern: 需要镜像的正则

    definition:

        ha-mode: 指明镜像队列的模式，有效值为 all/exactly/nodes

            all     表示在集群所有的节点上进行镜像，无需设置ha-params

            exactly 表示在指定个数的节点上进行镜像，节点的个数由ha-params指定

            nodes   表示在指定的节点上进行镜像，节点名称通过ha-params指定

        ha-params: ha-mode 模式需要用到的参数

        ha-sync-mode: 镜像队列中消息的同步方式，有效值为automatic，manually

    apply-to: 可选值3个，默认all

        exchanges 表示镜像 exchange (并不知道意义所在)

        queues    表示镜像 queue

        all       表示镜像 exchange和queue

eg:

./rabbitmqctl set\_policy test "test" '{"ha-mode":"all","ha-sync-mode":"automatic"}'

测试: exchange = test, queue = test

    case1: pattern=test, apply-to=exchanges -> 结果 exchange被镜像

    case2: pattern=test, apply-to=queues    -> 结果 queue被镜像

    case3: pattern=test, apply-to=all      -> 结果 queue被镜像

结论: 不知道exchange被镜像的意义所在，镜像queue才是关键

### 集群配置策略

    保证集群的高可用，至少要有1个disc节点  
    RabbitMQ Cluster 全部挂掉，RAM节点无法先启动，必须先启动disc节点  
    推荐 2 RAM 1 DISC 集群搭建方式。

在上面我们指定了 ha-mode 的值为 all ，代表消息会被同步到所有节点的相同队列中。这里我们之所以这样配置，因为我们本身只有三个节点，因此复制操作的性能开销比较小。如果你的集群有很多节点，那么此时复制的性能开销就比较大，此时需要选择合适的复制系数。通常可以遵循过半写原则，即对于一个节点数为 n 的集群，只需要同步到 n/2+1 个节点上即可。此时需要同时修改镜像策略为 exactly，并指定复制系数 ha-params，示例命令如下：

|  |  |
| --- | --- |
| 1 | rabbitmqctl set\_policy ha-two "^" '{"ha-mode":"exactly","ha-params":2,"ha-sync-mode":"automatic"}' |

除此之外，RabbitMQ 还支持使用正则表达式来过滤需要进行镜像操作的队列，示例如下：

|  |  |
| --- | --- |
|  | rabbitmqctl set\_policy ha-all "^ha\." '{"ha-mode":"all"}' |

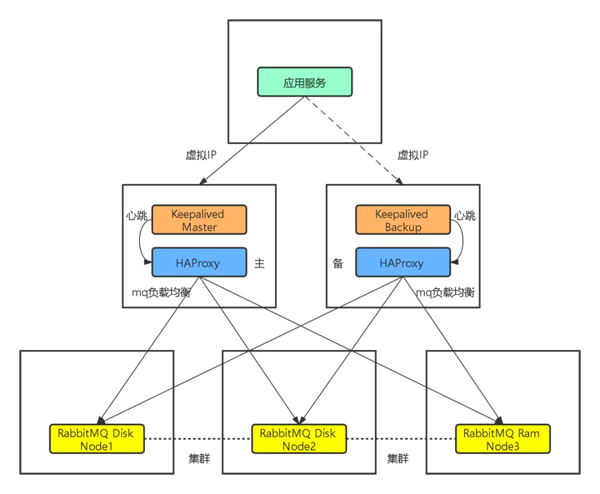
### 总结

    RabbitMQ高可用集群还是非常有必要的，高可用的代价就是性能的降低，对可靠性要求比较高的企业务还是值得的，据我测试2R1D镜像集群(非压测, 压测结果绝对更高)，达到1000QPS+还是没问题的，如果开启事务，保证同步发送应答，也可达500QPS+，绝对满足大多数可靠性要求高的业务。

## 快速搭建高可用RabbitMQ集群和HAProxy软负载

将两个 RabbitMQ 磁盘节点和一个 RabbitMQ 内存节点组成一个内建集群，之所以要用两个磁盘节点是防止，唯一的磁盘节点挂掉后，不能重建队列，交换器

**RabbitMQ 高可用集群架构：**

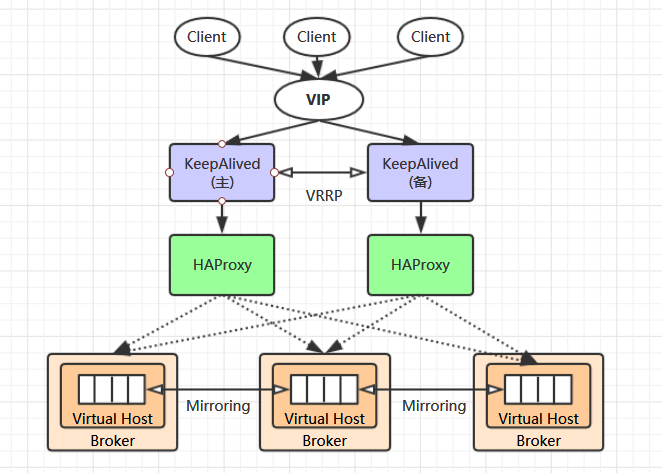


将两个 RabbitMQ 磁盘节点和一个 RabbitMQ 内存节点组成一个内建集群，之所以要用两个磁盘节点是防止，唯一的磁盘节点挂掉后，不能重建队列，交换器。用 HAProxy 作为 RabbitMQ 集群的负载均衡。为了防止 HAProxy 单点故障，用 Keepalived 将两个 HAProxy 节点做成一主一备。应用使用 VIP（虚拟IP） 访问 HAProxy 服务时，默认连接主机（Master）的 HAProxy，当主机（Master）上的 HAProxy 故障时，VIP 会漂移到备机（Backup）上，就会连接备机（Backup）上的 HAProxy 服务。

# 基于 HAProxy + KeepAlived 搭建 RabbitMQ 高可用集群

### 集群架构

当单台 RabbitMQ 服务器的处理消息的能力达到瓶颈时，此时可以通过 RabbitMQ 集群来进行扩展，从而达到提升吞吐量的目的。RabbitMQ 集群是一个或多个节点的逻辑分组，集群中的每个节点都是对等的，每个节点共享所有的用户，虚拟主机，队列，交换器，绑定关系，运行时参数和其他分布式状态等信息。一个高可用，负载均衡的 RabbitMQ 集群架构应类似下图：



这里对上面的集群架构做一下解释说明：

首先一个基本的 RabbitMQ 集群不是高可用的，虽然集群共享队列，但在默认情况下，消息只会被路由到某一个节点的符合条件的队列上，并不会同步到其他节点的相同队列上。假设消息路由到 node1 的 my-queue 队列上，但是 node1 突然宕机了，那么消息就会丢失，想要解决这个问题，需要开启队列镜像，将集群中的队列彼此之间进行镜像，此时消息就会被拷贝到处于同一个镜像分组中的所有队列上。

其次 RabbitMQ 集群本身并没有提供负载均衡的功能，也就是说对于一个三节点的集群，每个节点的负载可能都是不相同的，想要解决这个问题可以通过硬件负载均衡或者软件负载均衡的方式，这里我们选择使用 HAProxy 来进行负载均衡，当然也可以使用其他负载均衡中间件，如 LVS 等。HAProxy 同时支持四层和七层负载均衡，并基于单一进程的事件驱动模型，因此它可以支持非常高的井发连接数。

接着假设我们只采用一台 HAProxy ，那么它就存在明显的单点故障的问题，所以至少需要两台 HAProxy ，同时这两台 HAProxy 之间需要能够自动进行故障转移，通常的解决方案就是 KeepAlived 。KeepAlived 采用 VRRP (Virtual Router Redundancy Protocol，虚拟路由冗余协议) 来解决单点失效的问题，它通常由一组一备两个节点组成，同一时间内只有主节点会提供对外服务，并同时提供一个虚拟的 IP 地址 (Virtual Internet Protocol Address ，简称 VIP) 。 如果主节点故障，那么备份节点会自动接管 VIP 并成为新的主节点 ，直到原有的主节点恢复。

最后，任何想要连接到 RabbitMQ 集群的客户端只需要连接到虚拟 IP，而不必关心集群是何种架构，

1. 负载均衡分为L4 switch（四层交换），即在OSI第4层工作，就是TCP层啦。此种Load Balance不理解应用协议（如HTTP/FTP/MySQL等等）。例子：LVS，F5。

另一种叫做L7 switch（七层交换），OSI的最高层，应用层。此时，该Load Balancer能理解应用协议。例子： haproxy，MySQL Proxy。

# 附录

### 消费者中设置死信队列

package com.qiyexue.api.dlx;

import com.rabbitmq.client.Channel;

import com.rabbitmq.client.Connection;

import com.rabbitmq.client.ConnectionFactory;

import java.util.HashMap;

import java.util.Map;

/\*\*

\* 消费者

\*

\* @author 七夜雪

\* @date 2018-12-15 20:07

\*/

public class Consumer {

public static void main(String[] args) throws Exception {

// 1. 创建连接工厂并设置属性

ConnectionFactory factory = new ConnectionFactory();;

factory.setHost("192.168.72.138");

factory.setPort(5672);

factory.setVirtualHost("/");

// 2. 创建连接

Connection connection = factory.newConnection();

// 3. 创建channel

Channel channel = connection.createChannel();

// 4. 声明死信队列Exchange和Queue

channel.exchangeDeclare("dlx.exchange", "topic");

channel.queueDeclare("dlx.queue", true, false, false, null);

channel.queueBind("dlx.queue", "dlx.exchange", "#");

// 5. 声明普通Exchange

String exchangeName = "test\_dlx\_exchange";

String exchangeType = "topic";

String routingKey = "dlx.\*";

channel.exchangeDeclare(exchangeName, exchangeType, true, false, null);

// 6. 声明消息队列, 指定死信队列为dlx.exchange

String queueName = "test\_dlx\_queue";

Map<String, Object> arguments = new HashMap<>();

// x-dead-leeter-exchange属性用于指定死信队列为dlx.exchange

arguments.put("x-dead-letter-exchange", "dlx.exchange");

channel.queueDeclare(queueName, true, false, false, arguments);

// 6. 绑定队列和Exchange

channel.queueBind(queueName, exchangeName, routingKey);

}

}

### 生产者消息设置超时时间

package com.qiyexue.api.dlx;

import com.rabbitmq.client.\*;

import java.util.HashMap;

import java.util.Map;

/\*\*

\* 生产者

\*

\* @author 七夜雪

\* @date 2018-12-15 19:56

\*/

public class Producer {

public static void main(String[] args) throws Exception {

// 1. 创建ConnectionFactory, 并设置属性

ConnectionFactory factory = new ConnectionFactory();

factory.setHost("192.168.72.138");

factory.setPort(5672);

factory.setVirtualHost("/");

// 2. 创建连接

Connection connection = factory.newConnection();

// 3. 创建channel

Channel channel = connection.createChannel();

String exchangeName = "test\_dlx\_exchange";

String routingKey = "dlx.qiye";

AMQP.BasicProperties properties = new AMQP.BasicProperties().builder().expiration("5000").build();

// 发送消息

String msg = "Hello, 七夜雪";

channel.basicPublish(exchangeName, routingKey, true, properties, msg.getBytes());

// 关闭连接

channel.close();

connection.close();

}

}