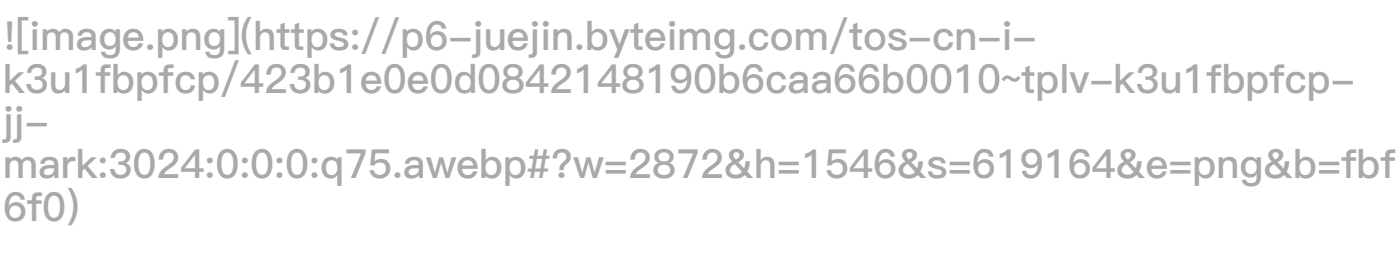
一文带大家快速掌握RabbitMQ!

前言

周末花了2天时间学习了RabbitMQ，总结了最核心的知识点，带大家快速掌握RabbitMQ，整理不易**希望帮忙点赞，转发，分享下哈，谢谢**

文章内容收录到个人网站，方便阅读：[hardyfish.top/](http://cxyroad.com/ "http://hardyfish.top/")



基础知识

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完全不同的应用之间共享数据，它是使用`Erlang`语言来编写的，并且是基于`AMQP`协议的；

RabbitMQ高性能的原因

* Erlang语言在交换机的交互方面性能优秀的（`Erlang`语言最初在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是非常优秀的）
* Erlang有着和原生`Socket`一样的延迟

安装教程：[www.rabbitmq.com/install-deb...](http://cxyroad.com/ "https://www.rabbitmq.com/install-debian.html")

AMQP协议

什么是AMQP高级消息队列协议

AMQP(Advanced Message Queueing Protocol)定义：具有现代特征的二进制协议。是一个提供统一消息服务的应用层标准高级消息队列协议，是应用层协议的一个开放标准，为面向消息的中间件设计

AMQP协议模型：

Publisher 推送消息前先与Server建立连接，找到`Virtual host`，然后将消息推送至Exchange交换机。而交换机与`Message Queue`有绑定关系(一个交换机相当于一个独立的虚拟机，而这个虚拟机内的各种独立的应用就相当于一个Queue，这个Queue与交换机绑定)，`Consumer`通过绑定的对队列，而交换机也绑定了队列。发送者将消息发送给交换机，这样就能完成消息的推送了

整体架构图

基本概念

Broker

消息队列服务进程，接收客户端的连接，实现AMQP实体服务。

Connection

连接，应用程序与`Broker`的网络连接。

Producer

消息生产者，即生产方客户端，生产方客户端将消息发送到MQ。

Consumer

消息消费者，即消费方客户端，接收MQ转发的消息。

Channel

网络信道，几乎所有的操作都在`Channel`中进行，Channel是进行消息读写的通道。客户端可建立多个Channel，每个Channel代表一个会话任务

Message

消息，服务器和应用程序之间传送的数据，由`Properties`和Body组成。Properties可以对消息进行修饰，比如消息的优先级、延迟等高级特性；Body则就是消息体内容。

Virtual Host

虚拟地址，用于进行逻辑隔离，最上层的消息路由。一个Virtual Host里面可以有若干个Exchange和Queue，同一个`Virtual Host`里面不能有相同名称的Exchange或Queue

Exchange

交换机，接收消息，根据路由键转发消息到绑定的队列。

24)

> 常见的有4种不同的交换机类型：

* 直连交换机：Direct exchange
* 扇形交换机：Fanout exchange
* 主题交换机：Topic exchange
* 首部交换机：Headers exchange

> 扇形交换机

扇形交换机是最基本的交换机类型，扇形交换机会把能接收到的消息全部发送给绑定在自己身上的队列。因为广播不需要思考，所以扇形交换机处理消息的速度也是所有的交换机类型里面最快的

> 直连交换机

直连交换机是一种带路由功能的交换机，一个队列会和一个交换机绑定，除此之外再绑定一个`routing_key`，当消息被发送的时候，需要指定一个`binding_key`，这个消息被送达交换机的时候，就会被这个交换机送到指定的队列里面去。同样的一个`binding_key`也是支持应用到多个队列中的

这样当一个交换机绑定多个队列，就会被送到对应的队列去处理

适用场景：有优先级的任务，根据任务的优先级把消息发送到对应的队列，这

样可以指派更多的资源去处理高优先级的队列

> 主题交换机

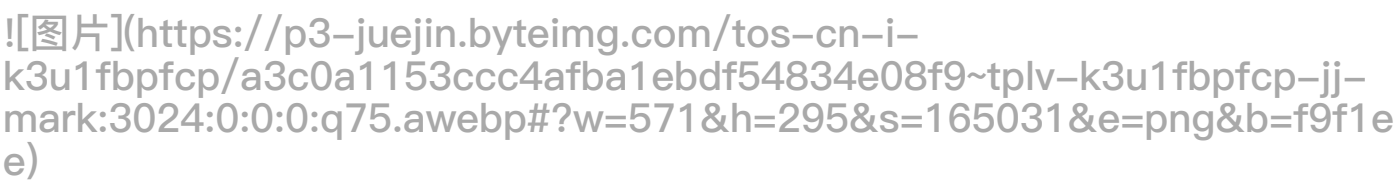
直连交换机的`routing_key`方案非常简单，如果我们希望一条消息发送给多个队列，那么这个交换机需要绑定上非常多的`routing_key`，假设每个交换机上都绑定一堆的`routing_key`连接到各个队列上。那么消息的管理就会异常地困难。

所以`RabbitMQ`提供了一种主题交换机，发送到主题交换机上的消息需要携带指定规则的`routing_key`，主题交换机会根据这个规则将数据发送到对应的(多个)队列上。

主题交换机的`routing_key`需要有一定的规则，交换机和队列的`binding_key`需要采用`*.#.*.....`的格式，每个部分用`.`分开，其中：

* `*`表示一个单词
* `#`表示任意数量（零个或多个）单词

当一个队列的绑定键为`#`的时候，这个队列将会无视消息的路由键，接收所有的消息



> 首部交换机

首部交换机是忽略`routing_key`的一种路由方式。路由器和交换机路由的规则是通过`Headers`信息来交换的，这个有点像`HTTP`的`Headers`。

将一个交换机声明成首部交换机，绑定一个队列的时候，定义一个`Hash`的数据结构，消息发送的时候，会携带一组hash数据结构的信息，当`Hash`的内容匹配上的时候，消息就会被写入队列。

绑定交换机和队列的时候，Hash结构中要求携带一个键`x-match`，这个键的Value可以是any或者all，这代表消息携带的Hash是需要全部匹配(all)，还是仅匹配一个键(any)就可以了

相比直连交换机，首部交换机的优势是匹配的规则不被限定为字符串

* any: 只要在发布消息时携带的有一对键值对headers满足队列定义的多个参数`arguments`的其中一个就能匹配上，注意这里是键值对的完全匹配，只匹配到键了，值却不一样是不行的；
* all：在发布消息时携带的所有`Entry`必须和绑定在队列上的所有Entry完全匹配

Binding

Exchange和Queue之间的虚拟连接，Exchange在与多个Message Queue发生Binding后会生成一张路由表，路由表中存储着`Message Queue`所需消息的限制条件即Binding Key。当Exchange收到Message时会解析其Header得到Routing Key，Exchange根据Routing Key与Exchange Type将Message路由到Message Queue。Binding Key由Consumer在`Binding Exchange`与Message Queue时指定，而Routing Key由Producer发送Message时指定，两者的匹配方式由Exchange Type决定

Routing Key

一个路由规则，虚拟机可用它来确定如何路由一个特定消息。

Queue

也称为`Message Queue`，消息队列，保存消息并将它们转发给消费者。

消息发布流程：

1. 生产者和Broker建立TCP连接。
2. 生产者和`Broker`建立通道。
3. 生产者通过通道消息发送给Broker，由Exchange将消息进行转发。
4. Exchange将消息转发到指定的Queue（队列）

消息接收流程：

1. 消费者和Broker建立TCP连接 。
2. 消费者和`Broker`建立通道。
3. 消费者监听指定的Queue（队列）
4. 当有消息到达Queue时Broker默认将消息推送给消费者。
5. 消费者接收到消息。

消息流转过程

生产者生产出Message并投递到`Exchange`上

一个Exchange可以绑定多个`Message Queue`，它根据路由策略（`routing key`）路由到指定的队列，最后由消费端去监听队列

工作模式

队列模式：

对于任务过重或任务较多情况使用工作队列可以提高任务处理的速度。

1、一条消息只会被一个消费者接收；

2、rabbitmq采用轮询的方式将消息是平均发送给消费者的；

3、消费者在处理完某条消息后，才会收到下一条消息。

发布订阅模式：

1、每个消费者监听自己的队列。

2、生产者将消息发给`broker`，由交换机将消息转发到绑定此交换机的每个队列，每个绑定交换机的队列都将接收到消息

对应交换机中的`fanout`类型

路由模式：

1、每个消费者监听自己的队列，并且设置routingkey。

2、生产者将消息发给交换机，由交换机根据`routingkey`来转发消息到指定的队列。

对应交换机中的`direct`类型

通配符模式：

对应交换机中的`topics`类型

Header转发器模式：

对应交换机中的`header`类型

远程过程调用模式：

RPC即客户端远程调用服务端的方法，使用MQ可以实现RPC的异步调用，基于Direct交换机实现，流程如下：

1. 客户端即是生产者就是消费者，向`RPC`请求队列发送RPC调用消息，同时监听RPC响应队列。
2. 服务端监听RPC请求队列的消息，收到消息后执行服务端的方法，得到方法返回的结果。
3. 服务端将RPC方法的结果发送到`RPC`响应队列。
4. 客户端（RPC调用方）监听RPC响应队列，接收到RPC调用结果。

基本使用
====

基本命令行操作
=======

关于服务的操作：

1. 服务启动：`rabbitmqctl start_app` / `rabbitmq-server start &`
2. 服务停止：`rabbitmqctl stop_app` / `rabbitmq-server stop`
3. 服务重启：`service rabbitmq-server restart`
4. 节点状态：`rabbitmqctl status`

关于用户的操作：

1. 添加用户：`rabbitmqctl add_user username password`
2. 列出所有用户：`rabbitmqctl list_users`
3. 删除用户：`rabbitmqctl delete_user username`
4. 清除用户权限：`rabbitmqctl clear_permissions -p vhostpath username`
5. 列出用户权限：`rabbitmqctl list_user_permissions username`
6. 修改密码：`rabbitmqctl change_password username newpassword`
7. 设置用户权限：`rabbitmqctl set_permissions -p vhostpath username ".*" ".*" ".*"`

关于虚拟主机的操作：

1. 创建虚拟主机：`rabbitmqctl add_vhost vhostpath`
2. 列出所有虚拟主机：`rabbitmqctl list_vhost`
3. 列出虚拟主机上所有权限：`rabbitmqctl list_permissions -p vhostpath`
4. 删除虚拟主机：`rabbitmqctl delete_vhost vhostpath`

关于消息队列的操作：

1. 查看所有队列信息：`rabbitmqctl list_queues`
2. 清除队列里的消息：`rabbitmqctl -p vhostpath purge_queue blue`

高级命令

1. `rabbitmqctl reset`：移除所有数据，要在`rabbitmqctl stop_app`之后使用
2. 组成集群命令：`rabbitmqctl join_cluster [--ram]` （ram内存级别存储，disc磁盘）
3. 查看集群状态：`rabbitmqctl cluster_status`
4. 修改集群节点的存储形式：`rabbitmqctl change_cluster_node_type disc | ram`
5. 忘记（摘除）节点：`rabbitmqctl forget_cluster_node [--offline]`（offline服务不启动的情况下）
6. 修改节点名称：`rabbitmqctl rename_cluster_node oldnode1 newnode1 [oldnode2] [newnode2 ...]`

> 集群配置失败，故障转移等情况下可以将启动失败的节点给移除掉。它可以在不启动的情况下对节点的摘除

入门使用

引入RabbitMQ依赖：

```
<dependency>
 <groupId>com.rabbitmq</groupId>
 <artifactId>amqp-client</artifactId>
 <version>3.6.5</version>
</dependency>
```

创建一个生产者

```
public class Producer {

    private static final String QUEUE_NAME = ”test01”;

    public static void main(String[] args) throws IOException, TimeoutException {
```

```
        // 1. 创建连接工厂并配置
        ConnectionFactory connectionFactory = new ConnectionFactory();
        connectionFactory.setHost("192.168.58.129");
        connectionFactory.setPort(5672);
        // 设置虚拟机
        connectionFactory.setVirtualHost("/test");
        // 2. 通过连接工厂建立连接
        Connection connection = connectionFactory.newConnection();

        // 3. 通过connection创建Channel
        Channel channel = connection.createChannel();

        // 4. 通过Channel发送数据 (exchange, routingKey, props, body)
        // 不指定Exchange时, 交换机默认是AMQP default, 此时就看
RoutingKey
        // RoutingKey要等于队列名才能被路由, 否则消息会被删除
        for (int i = 0; i < 5; i++) {
            String msg = "Learn For RabbitMQ-" + i;
            channel.basicPublish("", QUEUE_NAME, null, msg.getBytes());
            System.out.println("Send message : " + msg);
        }

        // 5.关闭连接
        channel.close();
        connection.close();
    }
}

```

创建一个消费者

```
public class Consumer {

    private static final String QUEUE_NAME = "test01";

    public static void main(String[] args) throws IOException,
TimeoutException {
        // 1. 创建连接工厂并配置
        ConnectionFactory connectionFactory = new ConnectionFactory();
        connectionFactory.setHost("192.168.58.129");
        connectionFactory.setPort(5672);
        // 设置虚拟机
        connectionFactory.setVirtualHost("/test");
        // 2. 通过连接工厂建立连接
```

```
        Connection connection = connectionFactory.newConnection();

        // 3. 通过connection创建Channel
        Channel channel = connection.createChannel();

        // 4. 声明队列 (queue, durable, exclusive, autoDelete, args)
        channel.queueDeclare(QUEUE_NAME, true, false, false, null);

        // 5. 创建消费者
        DefaultConsumer defaultConsumer = new
DefaultConsumer(channel) {
            /**
             * 获取消息 （监听到有消息时调用）
             * @param consumerTag 消费者标签, 在监听队列时可以设置
autoAck为false,即手动确认（避免消息的丢失）, 消息唯一性处理
             * @param envelope 信封
             * @param properties 消息的属性
             * @param body 消息的内容
             * @throws IOException
             */
            @Override
            public void handleDelivery(String consumerTag, Envelope
envelope, AMQP.BasicProperties properties, byte[] body) throws
IOException {
                String msg = new String(body, "utf-8");
                System.out.println("Received message : " + msg);
            }
        };

        // 6. 设置Channel, 监听队列(String queue, boolean
autoAck,Consumer callback)
        channel.basicConsume(QUEUE_NAME, true, defaultConsumer);


    }
}

```

参数：

* `queue`：队列名称
* `durable`：持久化，true 即使服务重启也不会被删除
* `exclusive`：独占，true 队列只能使用一个连接，连接断开队列删除
* `autoDelete`：自动删除，true 脱离了Exchange（连接断开），即队列没有Exchange关联时，自动删除
* `arguments`：扩展参数

* `autoAck`：是否自动签收（回执）

不指定Exchange时，交换机默认是AMQP default，此时就看`RoutingKey`，RoutingKey要等于队列名才能被路由，否则消息会被删除

交换机属性

`Name`：交换机名称

`Type`：交换机类型—— direct、topic、fanout、header

`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，即Exchange上没有队列绑定，自动删除该Exhcange

`Internal`：当前Exchange是否用于RabbitMQ内部使用，大多数使用默认False

`Arguments`：扩展参数，用于扩展AMQP协议定制化使用

> **Direct Exchange**：

```
 // Consumer
 // 声明交换机:
 // (String exchange, String type, boolean durable, boolean autoDelete, boolean internal, Map<String, Object) arguments)
 channel.exchangeDeclare(”exchangeName”, BuiltinExchangeType.DIRECT, true, false, false, null);

 // 声明队列 (String queue, boolean durable, boolean exclusive, boolean autoDelete, Map<String, Object) args)
 channel.queueDeclare(”queueName”, true, false, false, null);

 // 建立绑定关系:
```

```
 channel.queueBind(”queueName”, ”exchangeName”, ”routingKey”);
 //
==========
 // Producer
 // 发送消息 (String exchange, String routingKey, BasicProperties props,
Bytes[] body)
 channel.basicPublish(”exchangeName”, ”routingKey”, null,
”msg”.getBytes());

```

> **Topic Exchange**:

```
 // Consumer
 // 声明交换机:
 // (String exchange, String type, boolean durable, boolean autoDelete,
boolean internal, Map<String, Object) arguments)
 channel.exchangeDeclare(”exchangeName”,
BuiltinExchangeType.TOPIC, true, false, false, null);

 // 声明队列 (String queue, boolean durable, boolean exclusive, boolean
autoDelete, Map<String, Object) args)
 channel.queueDeclare(”queueName”, true, false, false, null);

 // 建立绑定关系:
 channel.queueBind(”queueName”, ”exchangeName”, ”routingKey.#”);
 //
==========
 // Producer
 // 发送消息 (String exchange, String routingKey, BasicProperties props,
Bytes[] body)
 channel.basicPublish(”exchangeName”, ”routingKey.hi”, null,
”msg”.getBytes());
 channel.basicPublish(”exchangeName”, ”routingKey.save”, null,
”msg”.getBytes());
 channel.basicPublish(”exchangeName”, ”routingKey.save.hi”, null,
”msg”.getBytes());

```

因为使用了模糊匹配的”`#`”，可以匹配到发送的三条消息。因此可以收到三条消息

> **Fanout Exchange**:

```
 // Consumer
 // 声明交换机:
 // (String exchange, String type, boolean durable, boolean autoDelete, boolean internal, Map<String, Object) arguments)
 channel.exchangeDeclare(”exchangeName”, BuiltinExchangeType.FANOUT, true, false, false, null);

 // 声明队列 (String queue, boolean durable, boolean exclusive, boolean autoDelete, Map<String, Object) args)
 channel.queueDeclare(”queueName”, true, false, false, null);

 // 建立绑定关系:
 // （不设置routingKey, 这里为空）
 channel.queueBind(”queueName”, ”exchangeName”, ””);
 //
==========
 // Producer
 // 发送消息 (String exchange, String routingKey, BasicProperties props, Bytes[] body)
 // 同样routingKey为空 （也可以是任意字符串, 因为fanout并不依据routingKey）
 channel.basicPublish(”exchangeName”, ””, null, ”msg”.getBytes());
```

高级特性
====

可靠性投递

什么是生产端的可靠性投递

* 保障消息的成功发出
* 保障MQ节点成功接收
* 发送端收到MQ节点（Broker）确认应答（已收到）

* 完善消息进行补偿机制

可靠性投递的方案一

消息落库（持久化至数据库），对消息状态进行打标，如若消息未响应，进行轮询操作

1.把业务消息落库，再生成一条消息落库到消息DB用来记录（譬如消息刚创建，正在发送中 status: 0）

> 缺点：对数据库进行两次持久化

2.生产端发送消息。

3.Broker端收到后，应答至生产端。`Confirm Listener`异步监听Broker的应答。

4.应答表明消息投递成功后，去消息DB中抓取到指定的消息记录，更新状态，如status: 1

5.如在3中出现网络不稳定等情况，导致Listener未收到消息成功确认的应答。

那么消息数据库中的status就还是0，而Broker可能是接收到消息的状态。

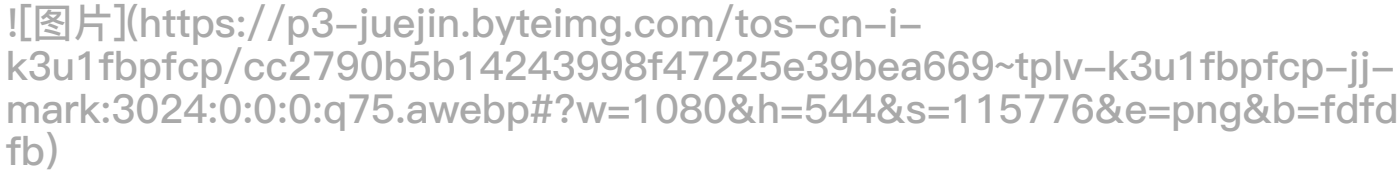
因此设定一个规则（定时任务），例如消息在落库5分钟后（超时）还是0的状态，就把该条记录抽取出来。

6.重新投递

7.限制一个重试的次数，譬如3次，如果大于3次，即为投递失败，更新status的值

> 用人工补偿机制去查询消息失败的原因

高并发场景消息的延迟投递，做二次确认，回调检查



Upstream service：生产端

Downstream service：消费端

1：业务消息落库后，发送消息至Broker。

2：紧接着发送第二条延迟（设置延迟时间）检查的消息。

3：消费端监听指定的队列接收到消息进行处理

4：处理完后，生成一条响应消息发送到Broker。

5：由Callback服务去监听该响应消息，收到该响应消息后持久化至消息DB（记录成功状态）。

6：到了延迟时间，延迟发送的消息也被Callback服务的监听器监听到后，去检查消息DB。如果未查询到成功的状态，Callback服务需要做补偿，发起RPC通讯，让生产端重新发送。生产端通过接收到的命令中所带的id去数据库查询该业务消息，再重新发送，即跳转到1。

该方案减少了对数据库的存储，保证了性能

消费端幂等性

避免消息的重复消费

消费端实现幂等性，接收到多条相同的消息，但不会重复消费，即收到多条一样的消息。

方案：

> 1.唯一ID + 指纹码机制

* 唯一ID + 指纹码（业务规则、时间戳等拼接）机制，利用数据库主键去重
* `SELECT COUNT(1) FROM T_ORDER WHERE ID = 唯一ID + 指纹码` 未查询到就`insert`，如有说明已处理过该消息，返回失败
* 优点：实现简单
* 缺点：高并发下有数据库写入的性能瓶颈
* 解决方案：根据ID进行分库分表、算法路由

> 2.利用Redis的原子性

需要考虑的问题：

* 是否要落库数据库，如落库，数据库和缓存如何做到数据的一致性
* 不落库，数据存储在缓存中，如何设置定时同步的策略（可靠性保障）

Confirm确认消息

生产者投递消息后，如果Broker收到消息，则会给生产者一个应答。

生产者进行接收应答，用来确认这条消息是否正常发送到Broker是消息可靠性投递的核心保障。

确认机制的流程图

发送消息与监听应答的消息是异步操作。

确认消息的实现

1. 在channel开启确认模式：`channel.confirmSelect();`
2. 在channel添加监听：`channel.addConfirmListener(ConfirmListener listener);` 返回监听成功和失败的结果，对具体结果进行相应的处理（重新发送、记录日志等待后续处理等）

具体代码：

```
public class ConfirmProducer {

    private static final String EXCHANGE_NAME = "confirm_exchange";
    private static final String ROUTING_KEY = "confirm.key";

    public static void main(String[] args) throws IOException, TimeoutException {

        ConnectionFactory connectionFactory = new ConnectionFactory();

        connectionFactory.setHost("192.168.58.129");
        connectionFactory.setPort(5672);
        connectionFactory.setVirtualHost("/test");

        Connection connection = connectionFactory.newConnection();

        Channel channel = connection.createChannel();

        // 指定消息的投递模式: 确认模式
        channel.confirmSelect();

        // 发送消息
```

```
        String msg = ”Send message of confirm demo”;
        channel.basicPublish(EXCHANGE_NAME, ROUTING_KEY, null,
msg.getBytes());

        // 添加确认监听
        channel.addConfirmListener(new ConfirmListener() {
            // 成功
            @Override
            public void handleAck(long deliveryTag, boolean multiple) throws
IOException {
                System.out.println(”========== Ack ========”);
            }

            // 失败
            @Override
            public void handleNack(long deliveryTag, boolean multiple)
throws IOException {
                System.out.println(”========== Nack ========”);
            }
        });
    }
}

```

Return消息机制

用于处理一些不可路由的消息。

基础API

有一个关键配置项：`Mandatory`：true，则监听器会接收到路由不可达的消息，然后进行处理；false，Broker会自动删除该消息

> 默认为false，当我们使用Return 消息机制的时候，我们需要将它设置为true

消息的生产者通过制定Exchange和`RoutingKey`，把消息投递到某一个队列中，消费者监听队列，进行消费。

但在一些情况下，发送消息时，Exchange不存在或`RoutingKey`路由不到

，Return Listener就会监听这种不可达的消息，然后进行处理。

Return Listener代码

```
public class ReturnProducer {

    private static final String EXCHANGE_NAME = "return_exchange";
    private static final String ROUTING_KEY = "return.key";
    private static final String ROUTING_KEY_ERROR = "wrong.key";

    public static void main(String[] args) throws IOException,
TimeoutException {

        ConnectionFactory connectionFactory = new ConnectionFactory();

        connectionFactory.setHost("192.168.58.129");
        connectionFactory.setPort(5672);
        connectionFactory.setVirtualHost("/test");

        Connection connection = connectionFactory.newConnection();

        Channel channel = connection.createChannel();


        // 消息
        String msg = "Send message of return demo";
        // 添加并设置Return监听器
        channel.addReturnListener(new ReturnListener() {
            @Override
            public void handleReturn(int replyCode, String replyText, String
exchange, String routingKey, AMQP.BasicProperties properties, byte[]
body) throws IOException {
                System.err.println("============ handleReturn
============");
                System.err.println("replyCode —— " + replyCode);
                System.err.println("replyText —— " + replyText);
                System.err.println("exchange —— " + exchange);
                System.err.println("routingKey —— " + routingKey);
                System.err.println("properties —— " + properties);
                System.err.println("body —— " + new String(body));
            }
        });

        // 设置Mandatory为true, 可以进行后续处理, 不会删除消息。
```

```
        // channel.basicPublish(EXCHANGE_NAME, ROUTING_KEY,
true,null, msg.getBytes());
        // 发送消息
        channel.basicPublish(EXCHANGE_NAME, ROUTING_KEY_ERROR,
true, null, msg.getBytes());

    }
}
```

消费端自定义监听

自定义监听的原因：

* 我们一般就是在代码中编写while循环，进行`consumer.nextDelivery`方法进行获取下一条消息，然后进行消费处理！
* 但是我们使用自定义的Consumer更加的方便，解耦性更加的强，也是在实际工作中最常用的使用方式！
* 非常简单，消费者只需要继承DefaultConsumer类，然后重写handleDelivery方法即可；

> 继承DefaultConsumer的此类被写出后，需要进行绑定。(在交换机绑定时绑定自定义的Consumer)；

```
public class ReturnConsumer {

    private static final String EXCHANGE_NAME = "return_exchange";
    private static final String ROUTING_KEY = "return.#";
    private static final String QUEUE_NAME = "return_queue";


    public static void main(String[] args) throws IOException,
TimeoutException {

        ConnectionFactory connectionFactory = new ConnectionFactory();

        connectionFactory.setHost("192.168.58.129");
        connectionFactory.setPort(5672);
        connectionFactory.setVirtualHost("/test");
```

```
        Connection connection = connectionFactory.newConnection();

        Channel channel = connection.createChannel();

        // 绑定交换机与队列, 指定路由键
        channel.exchangeDeclare(EXCHANGE_NAME,
BuiltinExchangeType.TOPIC, true);
        channel.queueDeclare(QUEUE_NAME, true, false, false, null);
        channel.queueBind(QUEUE_NAME, EXCHANGE_NAME,
ROUTING_KEY);

        DefaultConsumer defaultConsumer = new
DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag, Envelope
envelope, AMQP.BasicProperties properties, byte[] body) throws
IOException {
                System.out.println("Receive Message —— " + new
String(body));
            }
        };

        channel.basicConsume(QUEUE_NAME, true, defaultConsumer);
    }
}
```

消费端限流

当巨量消息瞬间全部推送时，单个客户端无法同时处理这些数据，服务器容易故障。因此要进行消费端限流

RabbitMQ提供了一种**Qos（服务质量保证）功能，即在非自动确认**前提下，如果一定数目的消息未被确认前（通过consume或者channel设置Qos值），不进行消费新消息。

```
void basicQos(int prefetchSize, int prefetchCount, boolean global)
throws IOException;
```

prefetchSize：消息限制大小，一般为0，不做限制。

prefetchCount：一次处理消息的个数，一般设置为1

global：一般为false。true，在channel级别做限制；false，在consumer级别做限制

```
public class QosConsumer {

    private static final String EXCHANGE_NAME = ”qos_exchange”;
    private static final String ROUTING_KEY = ”qos.#”;
    private static final String QUEUE_NAME = ”qos_queue”;


    public static void main(String[] args) throws IOException,
TimeoutException {

        ConnectionFactory connectionFactory = new ConnectionFactory();

        connectionFactory.setHost(”192.168.58.129”);
        connectionFactory.setPort(5672);
        connectionFactory.setVirtualHost(”/test”);
        connectionFactory.setUsername(”orcas”);
        connectionFactory.setPassword(”1224”);

        Connection connection = connectionFactory.newConnection();

        Channel channel = connection.createChannel();

        // 绑定交换机与队列, 指定路由键
        channel.exchangeDeclare(EXCHANGE_NAME,
BuiltinExchangeType.TOPIC, true);
        channel.queueDeclare(QUEUE_NAME, true, false, false, null);
        channel.queueBind(QUEUE_NAME, EXCHANGE_NAME,
ROUTING_KEY);

        DefaultConsumer defaultConsumer = new
DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag, Envelope
envelope, AMQP.BasicProperties properties, byte[] body) throws
IOException {
                System.out.println(”Receive Message —— ” + new
```

```
String(body));
                // 手动ack签收
                channel.basicAck(envelope.getDeliveryTag(), false); // 不批量
签收
            }
        };

        /**
         *  prefetchSize: 0 不限制消息大小
         *  prefetchCount: 一次处理消息的个数, ack后继续推送
         *  global: false 应用在consumer级别
         */
        channel.basicQos(0, 1, false);
        //限流:autoAck需设置为false, 关闭自动签收
        channel.basicConsume(QUEUE_NAME, false, defaultConsumer);
    }
}
```

限流需要设置`channel.basicQos(0, 1, false);`

关闭autoAck，且需要手动签收。

在重写的handleDelivery方法中，如果没有进行手动签收`channel.basicAck()`，那么消费端在接收消息时，因为prefetchCount设置为1，只会接收1条消息，剩下的消息的等待中，并不会被推送，直到手动ack后。

队列
--

消费端ACK与重回队列机制

> 消费端的手工ACK和NACK：

消费端进行消费时，可能由于业务异常，会调用NACK拒绝确认，而到了一定次数，就直接ACK，将异常消息进行日志的记录，然后进行补偿。

由于服务器宕机等严重问题，消费端没消费成功，重发消息后，需要手工

ACK保障消费端消费成功。

> 消费端的重回队列：

将没有处理成功的消息重新回递给Broker。

一般在实际应用中，会关闭重回队列。

TTL队列

TTL：Time To Live，生存时间。

可以指定消息的过期时间。

可以指定队列的过期时间，从消息入队列开始计算，超过了队列的超时时间设置，那么消息会自动清除

```
AMQP.BasicProperties properties = new AMQP.BasicProperties.Builder()
                    .deliveryMode(2)
                    .expiration(”10000”)
                    .build();
```

死信队列

DLX：Dead-Letter-Exchange

当消息在队列中变成死信时，能被重新publish到另一个Exchange，该Exchange就是DLX

> 发生死信队列的情况：

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

> 死信队列的设置：

1. 正常声明交换机，队列并绑定，需要在队列上设置一个参数：`arguments.put(”x-dead-letter-exchange”, ”dlx.exchange”);`
2. 声明死信队列的Exchange和Queue，然后进行绑定：`Exchange: dlx.exchange``Queue: dlx.queue``RoutingKey: #`
3. 在消息过期、requeue、队列达到最大长度时（即为死信），消息会发送到指定的`dlx.exchange`交换机上，消费者会监听该交换机所绑定的死信队列。

```
public class DlxConsumer {

    private static final String EXCHANGE_NAME = ”dlx_exchange”;
    private static final String ROUTING_KEY = ”dlx.#”;
    private static final String QUEUE_NAME = ”dlx_queue”;

 // DLX
    private static final String DLX_EXCHANGE = ”dlx.exchange”;
    private static final String DLX_QUEUE = ”dlx.queue”;
    private static final String DLX_ROUTING_KEY = ”#”;


    public static void main(String[] args) throws IOException,
TimeoutException {

        ConnectionFactory connectionFactory = new ConnectionFactory();

        connectionFactory.setHost(”192.168.58.129”);
        connectionFactory.setPort(5672);
        connectionFactory.setVirtualHost(”/test”);
        connectionFactory.setUsername(”orcas”);
        connectionFactory.setPassword(”1224”);

        Connection connection = connectionFactory.newConnection();

        Channel channel = connection.createChannel();
```

```
        channel.exchangeDeclare(EXCHANGE_NAME,
BuiltinExchangeType.TOPIC, true);
        // 1. 设置死信队列的参数
        Map<String, Object> arguments = new HashMap<>();
        arguments.put("x-dead-letter-exchange", DLX_EXCHANGE);
        channel.queueDeclare(QUEUE_NAME, true, false, false, arguments);
        channel.queueBind(QUEUE_NAME, EXCHANGE_NAME,
ROUTING_KEY);

        // 2. 声明死信队列
        channel.exchangeDeclare(DLX_EXCHANGE,
BuiltinExchangeType.TOPIC, true, false, null);
        channel.queueDeclare(DLX_QUEUE, true, false, false, null);
        channel.queueBind(DLX_QUEUE, DLX_EXCHANGE,
DLX_ROUTING_KEY);

        DefaultConsumer defaultConsumer = new
DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag, Envelope
envelope, AMQP.BasicProperties properties, byte[] body) throws
IOException {
                System.out.println("Receive Message —— " + new
String(body));
                // 手动ack签收
                channel.basicAck(envelope.getDeliveryTag(), false); // false
不批量签收
            }
        };
        channel.basicConsume(QUEUE_NAME, false, defaultConsumer);
    }
}

```

最后
==

**文章内容收录到个人网站，方便阅读
**：[hardyfish.top/](http://cxyroad.com/ "http://hardyfish.top/")

参考和鸣谢：

官网：[www.rabbitmq.com/](http://cxyroad.com/ "https://www.rabbitmq.com/")

视频：RabbitMQ消息中间件技术精讲

原文链接: https://juejin.cn/post/7356894848518389779