

RabbitMQ

RabbitMQ

MQ概念

定义

为什么使用MQ

MQ的缺点

RabbitMQ-概念

简介

四大核心概念

核心部分

名词介绍

特点

RabbitMQ-安装

安装和启动

Web管理界面及授权操作

RabbitMQ-简单案例

hello world分析

导入依赖

消息生产者

消息消费者

RabbitMQ-Work Queues

轮训分发消息

抽取工具类

消费者

生产者

运行结果

消息应答

自动应答

消息应答的方法

消息自动重新入队

消息手动应答代码

队列持久化

队列如何实现持久化？

消息持久化

不公平分发

预取值分发

RabbitMQ-发布确认

发布确认原理

发布确认的策略

开启发布确认的方法

单个确认发布

批量确认发布

异步确认发布

如何处理异步未确认消息

三种发布确认策略速度对比

RabbitMQ-交换机

Exchanges

概念

Exchanges的类型

无名Exchange

临时队列

绑定(bindings)

Fanout exchange

- 介绍
- 实现
- Direct exchange
 - 介绍
 - 多重绑定
 - 实现
- Topics exchange
 - 介绍
 - Topic的要求
 - 实现
- RabbitMQ-死信队列
 - 死信的概念
 - 死信的来源
 - 死信的实现
 - TTL
 - 消费者1
 - 生产者
 - 消费者2
 - 队列达到最大长度
 - 消息被拒
- RabbitMQ-延迟队列
 - 延迟队列的概念
 - 延迟队列的使用场景
 - RabbitMQ中的TTL
 - 整合springboot
 - 队列TTL
 - 代码架构图
 - 配置类
 - 生产者
 - 消费者
 - 延迟队列TTL优化
- RabbitMQ插件实现延迟队列
 - 安装延时队列插件
 - 配置类
 - 生产者
 - 消费者
- 总结
- RabbitMQ-发布确认高级
 - 发布确认springboot版本
 - 实现
 - 配置类
 - 消息生产者的回调接口
 - 生产者
 - 消费者
 - 回退消息
 - 备份交换机
 - 配置类
 - 报警消费者
- RabbitMQ-幂等性、优先级、惰性
 - 幂等性
 - 概念
 - 消息重复消费
 - 解决思路
 - 消费端的幂等性保障
 - 优先级队列
 - 使用场景
 - 如何添加
 - 惰性队列

MQ概念

定义

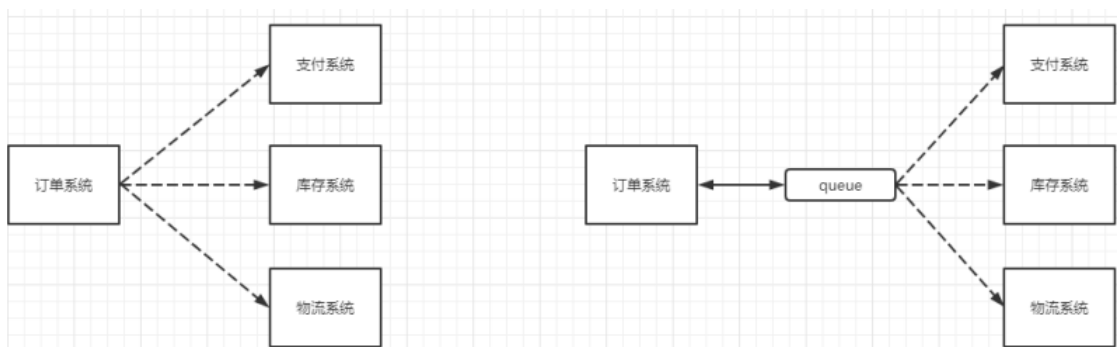
MQ(message queue), 从字面意思上看, 本质是个队列, FIFO 先入先出, 只不过队列中存放的内容是 message 而已, 还是一种跨进程的通信机制, 用于上下游

传递消息。在互联网架构中, MQ 是一种非常常见的上下游“逻辑解耦+物理解耦”的消息通信服务。使用了 MQ 之后, 消息发送上游只需要依赖 MQ, 不用依赖其

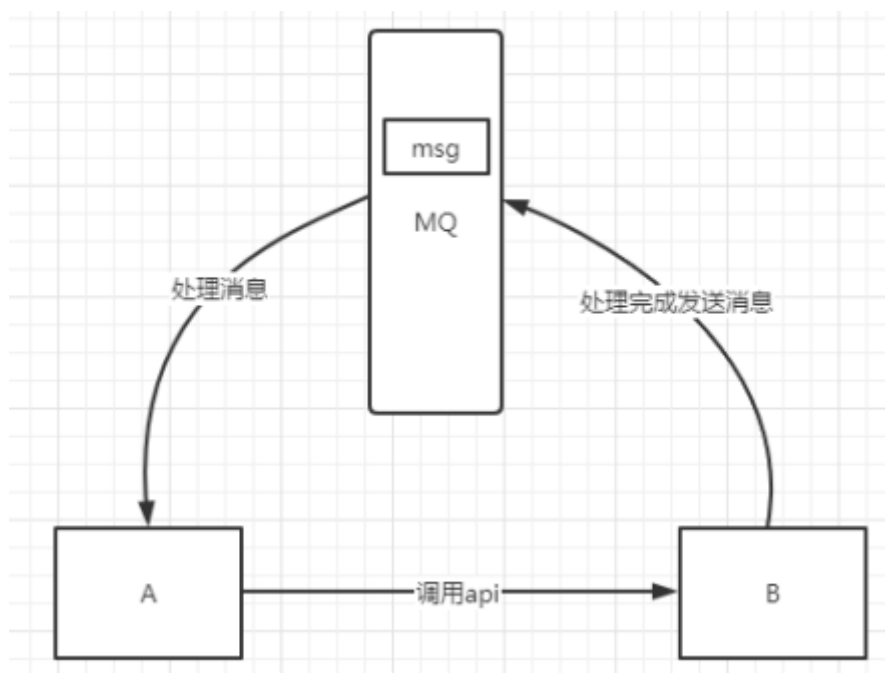
他服务。

为什么使用MQ

- 应用解耦: 将各个应用解耦, 只需从消息中订阅, 防止一处变动多处跟随修改



- 流量消峰: 并发量高峰期的时候, 可以从消息队列中稍后读取消息, 不会造成系统崩溃
- 异步处理: 发送接收双方不必同时在线, 加快响应速度



MQ的缺点

- 可用性降低：系统外部依赖变多，容易崩溃
- 复杂性提高：需要考虑消息的一致性问题、可靠传输问题、不被重复消费等问题

RabbitMQ-概念

简介

RabbitMQ 是一个消息中间件：它接受并转发消息。你可以把它当做一个快递站点，当你要发送一个包裹时，你把你的包裹放到快递站，快递员最终会把你的快递

送到收件人那里，按照这种逻辑 RabbitMQ 是一个快递站，一个快递员帮你传递快件。RabbitMQ 与快递站的主要区别在于，它不处理快件而是接收，存储和转

发消息数据。

RabbitMQ是一套开源的消息队列服务，基于 AMQP 的开源实现，由 Erlang 写成。

AMQP：Advanced Message Queue，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。

四大核心概念

- 生产者

产生数据发送消息的程序是生产者

- 交换机

交换机是 RabbitMQ 非常重要的一个部件，一方面它接收来自生产者的消息，另一方面它将消息推送到队列中。交换机必须确切知道如何处理它接收到的消

息，是将这些消息推送到特定队列还是推送到多个队列，亦或者是把消息丢弃，这个得有交换机类型决定

- 队列

队列是 RabbitMQ 内部使用的一种数据结构，尽管消息流经 RabbitMQ 和应用程序，但它们只能存储在队列中。队列仅受主机的内存和磁盘限制的约束，本

质上是一个大的消息缓冲区。许多生产者可以将消息发送到一个队列，许多消费者可以尝试从一个队列接收数据。这就是我们使用队列的方式

- 消费者

消费与接收具有相似的含义。消费者大多时候是一个等待接收消息的程序。请注意生产者，消费者和消息中间件很多时候并不在同一机器上。同一个应用程序

既可以是生产者又是可以是消费者。

核心部分

1 "Hello World!"

The simplest thing that does *something*



- [Python](#)
- [Java](#)

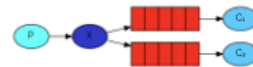
2 Work queues

Distributing tasks among workers (the [competing consumers pattern](#))



3 Publish/Subscribe

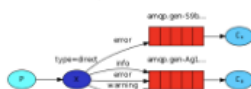
Sending messages to many consumers at once



- [Python](#)
- [Java](#)

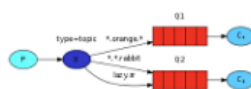
4 Routing

Receiving messages selectively



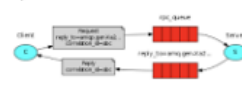
5 Topics

Receiving messages based on a pattern (topics)



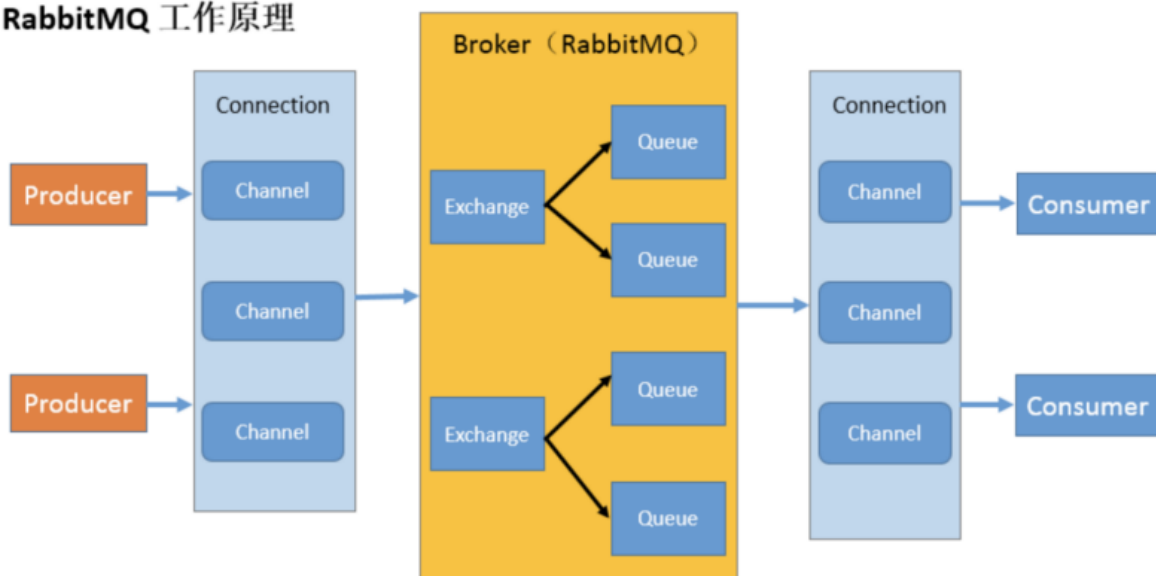
6 Publisher Confirms

Reliable publishing with publisher confirms



名词介绍

RabbitMQ 工作原理



- Broker

接收和分发消息的应用，RabbitMQ Server 就是 Message Broker

- Virtual host

出于多租户和安全因素设计的，把 AMQP 的基本组件划分到一个虚拟的分组中，类似于网络中的 namespace 概念。当多个不同的用户使用同一个

RabbitMQ server 提供的服务时，可以划分出多个 vhost，每个用户在自己的 vhost 创建 exchange / queue 等

- Connection

publisher / consumer 和 broker 之间的 TCP 连接

- Channel

如果每一次访问 RabbitMQ 都建立一个 Connection，在消息量大的时候建立 TCP Connection 的开销将是巨大的，效率也较低。Channel 是在 connection

内部建立的逻辑连接，如果应用程序支持多线程，通常每个 thread 创建单独的 channel 进行通讯，AMQP method 包含了 channel id 帮助客户端和

message broker 识别 channel，所以 channel 之间是完全隔离的。Channel 作为轻量级的 Connection 极大减少了操作系统建立 TCP connection 的开销

- Exchange

message 到达 broker 的第一站，根据分发规则，匹配查询表中的 routing key，分发 消息到 queue 中去。常用的类型有：direct (point-to-point), topic

(publish-subscribe) and fanout (multicast)

- Queue

消息最终被送到这里等待 consumer 取走

- Binding

exchange 和 queue 之间的虚拟连接，binding 中可以包含 routing key，Binding 信息被保存 到 exchange 中的查询表中，用于 message 的分发依据

特点

1. 可靠性 (Reliability)

RabbitMQ 使用多种机制来保证可靠性，如持久化、传输确认、发布确认。

2. 灵活的路由 (Flexible Routing)

在消息进入队列之前，通过 Exchange 来分配路由消息。对于典型的路由功能，RabbitMQ 已经提供了一些内置的 Exchange 来实现。针对更复杂的路由功

3. 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。

4. 消息集群 (Clustering)

在相同局域网中的多个RabbitMQ服务器可以聚合在一起，作为一个独立的逻辑代理来使用。

5. 高可用 (Highly Available 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。

6. 多协议 (Multi-protocol)

RabbitMQ 支持多种消息队列协议，比如 STOMP、MQTT 等等。

7. 多语言客户端 (Many Clients)

RabbitMQ 几乎支持所有常用语言，比如 Java、.NET、Ruby 等等。

8. 可视化管理工具 (Management UI)

RabbitMQ 提供了一个易用的用户界面，使得用户可以监控和管理消息代理的每一个环节。

9. 跟踪机制 (Tracing)

如果你的消息系统有异常行为，RabbitMQ还提供了追踪的支持，让你能够发现问题所在。

10. 插件机制 (Plugin System)

RabbitMQ附带了各种各样的插件来对自己进行扩展。你甚至也可以写自己的插件来使用。

RabbitMQ-安装

安装和启动

- 1、下载

官网下载地址：<https://www.rabbitmq.com/download.html>(opens new window)

这里我们选择的版本号（注意这两版本要求）

- rabbitmq-server-3.8.8-1.el7.noarch.rpm

GitHub: <https://github.com/rabbitmq/rabbitmq-server/releases/tag/v3.8.8>(opens new window)

加载下载: <https://packagecloud.io/rabbitmq/rabbitmq-server/packages/el/7/rabbitmq-server-3.8.8-1.el7.noarch.rpm>(opens new window)

- erlang-21.3.8.21-1.el7.x86_64.rpm

官网: <https://www.erlang-solutions.com/downloads/>

加速: https://packagecloud.io/rabbitmq/erlang/packages/el/7/erlang-21.3.8.21-1.el7.x86_64.rpm(opens new window)

注: Red Hat 8, CentOS 8 和 modern Fedora 版本, 把 “el7” 替换成 “el8”

2、安装

上传到个人安装目录

直接粘贴下面三行命令

```
1 rpm -ivh erlang-21.3.8.21-1.el7.x86_64.rpm
2 yum install socat -y
3 rpm -ivh rabbitmq-server-3.8.8-1.el7.noarch.rpm
```

3、启动

```
1 // 启动服务
2 systemctl start rabbitmq-server
3 // 查看服务状态
4 systemctl status rabbitmq-server
5 // 开机自启动
6 systemctl enable rabbitmq-server
7 // 停止服务
8 systemctl stop rabbitmq-server
9 // 重启服务
10 systemctl restart rabbitmq-server
```

Web管理界面及授权操作

1、安装

默认情况下, 是没有安装web端的客户端插件, 需要安装才可以生效

```
1 rabbitmq-plugins enable rabbitmq_management
```

安装完毕以后, 重启服务

```
1 systemctl restart rabbitmq-server
```

访问 <http://ip>地址:15672 , 用默认账号密码(guest)登录, 会出现权限问题

默认情况只能在 localhost 本机下访问, 所以需要添加一个远程登录的用户

2、添加用户

```

1 // 创建账号和密码
2 rabbitmqctl add_user admin 123456
3
4 // 设置用户角色
5 rabbitmqctl set_user_tags admin administrator
6
7 // 设置用户权限
8 // set_permissions [-p <vhostpath>] <user> <conf> <write> <read>
9 rabbitmqctl set_permissions -p "/" admin ".*" ".*" ".*"
10 // 用户 user_admin 具有/vhost1 这个 virtual host 中所有资源的配置、写、读权限
11
12 // 显示当前用户和角色
13 rabbitmqctl list_users

```

用户级别：

1. administrator：可以登录控制台、查看所有信息、可以对 rabbitmq 进行管理
2. monitoring：监控者 登录控制台，查看所有信息
3. policymaker：策略制定者 登录控制台，指定策略
4. managment：普通管理员 登录控制台

再次登录，用 admin 用户

3、重置命令

```

1 // 关闭应用的命令
2 rabbitmqctl stop_app
3
4 // 清除的命令为
5 rabbitmqctl reset
6
7 // 重新启动命令为
8 rabbitmqctl start_app

```

RabbitMQ-简单案例

hello world分析

我们将用 Java 编写两个程序。发送单个消息的生产者和接收消息并打印出来的消费者

在下图中，“P”是我们的生产者，“C”是我们的消费者。中间的框是一个队列 RabbitMQ 代表使用者保留的消息缓冲区



注意：连接的时候，需要开启 5672 端口

第一步：生产者代码

第二步：消费者代码



导入依赖

```
1  <!--指定 jdk 编译版本-->
2      <build>
3          <plugins>
4              <plugin>
5                  <groupId>org.apache.maven.plugins</groupId>
6                  <artifactId>maven-compiler-plugin</artifactId>
7                  <configuration>
8                      <source>8</source>
9                      <target>8</target>
10                 </configuration>
11             </plugin>
12         </plugins>
13     </build>
14
15     <dependencies>
16         <!--rabbitmq 依赖客户端-->
17         <dependency>
18             <groupId>com.rabbitmq</groupId>
19             <artifactId>amqp-client</artifactId>
20             <version>5.8.0</version>
21         </dependency>
22         <!--操作文件流的一个依赖-->
23         <dependency>
24             <groupId>commons-io</groupId>
25             <artifactId>commons-io</artifactId>
26             <version>2.6</version>
27         </dependency>
28     </dependencies>
```

消息生产者

```
1  import com.rabbitmq.client.Channel;
2  import com.rabbitmq.client.Connection;
3  import com.rabbitmq.client.ConnectionFactory;
4  /**
5   * @Author: ww
6   * @DateTime: 2022/4/25 22:30
7   * @Description: 生产者: 发消息
8   */
9  public class Producer {
10      // 队列名称
11      public static final String QUEUE_NAME = "hello";
12  }
```

```

13 // 发消息
14 public static void main(String[] args) throws Exception{
15     // 创建一个连接工厂
16     ConnectionFactory factory = new ConnectionFactory();
17     // 工厂ip 连接RabbitMQ的队列
18     factory.setHost("114.115.177.60");
19     // 用户名
20     factory.setUsername("admin");
21     // 密码
22     factory.setPassword("123");
23
24     // 创建连接
25     Connection connection = factory.newConnection();
26     // 获取信道
27     Channel channel = connection.createChannel();
28
29     /**
30      * 生成一个队列
31      * 参数1: 队列名称
32      * 参数2: 队列里面的消息是否持久化(true磁盘)，默认情况消息存储在内存中(false)
33      * 参数3: 该队列是否只供一个消费者进行消费(消息共享)，true可以多个消费者消费，
false只能一个消费者消费
34      * 参数4: 是否自动删除，最后一个消费者断开连接以后，该队列是否自动删除，true是，
false不是
35      * 参数5: 其它参数
36      */
37     channel.queueDeclare(QUEUE_NAME, false, false, false, null);
38     //发消息
39     String message = "hello world";
40     /**
41      * 发送一个消息
42      * 参数1: 送到哪个交换机
43      * 参数2: 路由的key值是哪个，本次是队列名称
44      * 参数3: 其它的参数信息
45      * 参数4: 发送消息的消息体
46      */
47     channel.basicPublish("", QUEUE_NAME, null, message.getBytes());
48     System.out.println("消息发送完毕");
49 }
50 }

```

消息消费者

```

1 import com.rabbitmq.client.*;
2
3 /**
4  * @Author: ww
5  * @DateTime: 2022/4/25 23:01
6  * @Description: 消费者 接收消息
7  */
8 public class Consumer {
9     // 队列名称
10     public static final String QUEUE_NAME = "hello";
11
12     // 接收消息

```

```

13     public static void main(String[] args) throws Exception{
14         // 创建一个连接工厂
15         ConnectionFactory factory = new ConnectionFactory();
16         // 工厂ip 连接RabbitMQ的队列
17         factory.setHost("114.115.177.60");
18         // 用户名
19         factory.setUsername("admin");
20         // 密码
21         factory.setPassword("123");
22         // 创建连接
23         Connection connection = factory.newConnection();
24         // 获取信道
25         Channel channel = connection.createChannel();
26
27         // 声明 接收消息
28         DeliverCallback deliverCallback = (consumerTag,message)->{
29             System.out.println(new String(message.getBody()));
30         };
31         // 声明 取消消息时的回调
32         CancelCallback cancelCallback = consumerTag->{
33             System.out.println("接收消息被中断");
34         };
35
36         /**
37          * 消费者接收消息
38          * 参数1: 接收哪个队列
39          * 参数2: 接收成功之后是否要自动应答, true:自动 false:手动
40          * 参数3: 消费者成功接收的回调
41          * 参数4: 消费者取消接收的回调
42          */
43
44         channel.basicConsume(QUEUE_NAME,true,deliverCallback,cancelCallback);
45     }

```

运行结果:

RabbitMQ-Work Queues

工作队列(又称任务队列)的主要思想是避免立即执行资源密集型任务,而不得不等待它完成。相反我们安排任务在之后执行。我们把任务封装为消息并将其发送到

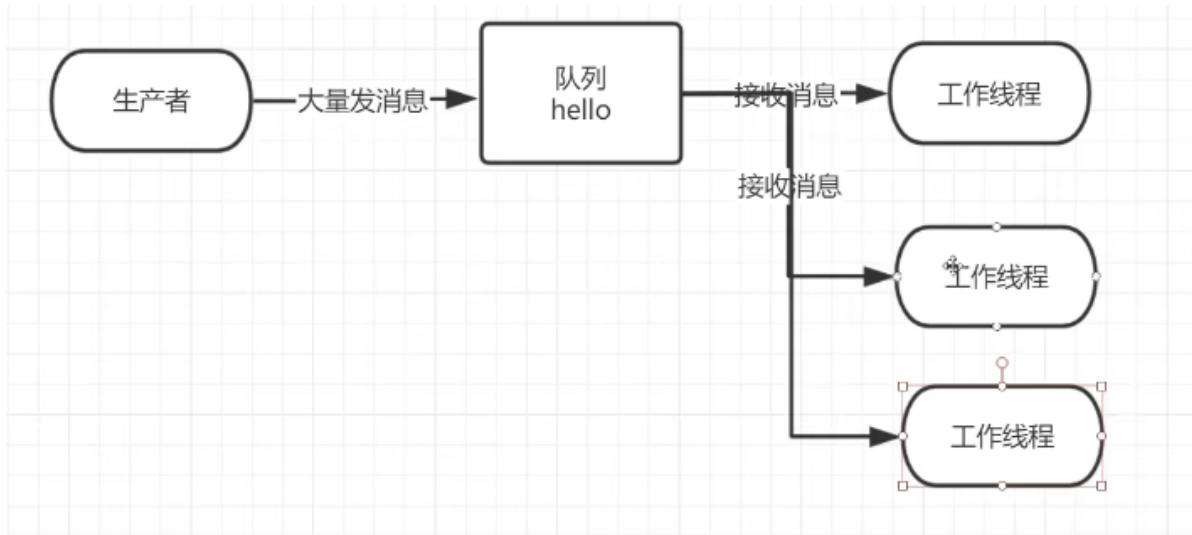
队列。在后台运行的工作进程将弹出任务并最终执行作业。当有多个工作线程时,这些工作线程将一起处理这些任务。

一个生产者,多个消费者,一个消息,只能被一个消费者获取,轮训分发消息,可以认为是simple模式的加强版。

当生产者生产速度较快,消费者不能及时消费时,就应该采用该模式。通常C1和C2获取到的消息不会重复,但rabbitmq本身设计,使其存在竞争风险。

资源竞争风险:由于rabbitmq中存储的任务数据,当存在ack_late时,任务没有执行完毕,队列中的消息不会删除,则会存在一个消息被多个消费者消费的危险。

解决方法：设置同步锁保证一个消息只能被一个worker消费。



轮训分发消息

抽取工具类

```
1 import com.rabbitmq.client.Channel;
2 import com.rabbitmq.client.Connection;
3 import com.rabbitmq.client.ConnectionFactory;
4
5 /**
6  * @Author: ww
7  * @DateTime: 2022/4/26 21:07
8  * @Description: 获取连接工厂创建信道的工具类
9  */
10 public class RabbitMqUtils {
11     //得到一个连接的 channel
12     public static Channel getChannel() throws Exception{
13         //创建一个连接工厂
14         ConnectionFactory factory = new ConnectionFactory();
15         factory.setHost("114.115.177.60");
16         factory.setUsername("admin");
17         factory.setPassword("123");
18         Connection connection = factory.newConnection();
19         Channel channel = connection.createChannel();
20         return channel;
21     }
22 }
```

消费者

```
1 import com.itww.rabbitmq.utils.RabbitMqUtils;
2 import com.rabbitmq.client.CancelCallback;
3 import com.rabbitmq.client.Channel;
4 import com.rabbitmq.client.DeliverCallback;
5
6 /**
7  * @Author: ww
8  * @DateTime: 2022/4/26 21:10
9  * @Description: 工作线程，相当于之前的消费者
10  */
```

```

11 public class Worker01 {
12
13     // 队列名称
14     public static final String QUEUE_NAME = "hello";
15
16     // 接收消息
17     public static void main(String[] args) throws Exception{
18         Channel channel = RabbitMQUtils.getChannel();
19
20         // 消息的接收
21         DeliverCallback deliverCallback = (consumerTag, message)->{
22             System.out.println("接收到的消息: " + new
String(message.getBody()));
23         };
24
25         // 消息接收被取消时
26         CancelCallback cancelCallback = consumerTag -> {
27             System.out.println("消息被取消");
28         };
29
30         /**
31          * 消费者接收消息
32          * 参数1: 接收哪个队列
33          * 参数2: 接收成功之后是否要自动应答, true:自动 false:手动
34          * 参数3: 消费者成功接收的回调
35          * 参数4: 消费者取消接收的回调
36          */
37         System.out.println("c2等待接收消息");
38
39         channel.basicConsume(QUEUE_NAME, true, deliverCallback, cancelCallback);
40     }
41 }

```

生产者

```

1 import com.rabbitmq.client.Channel;
2
3 import java.util.Scanner;
4
5 /**
6  * @Author: ww
7  * @DateTime: 2022/4/26 21:24
8  * @Description: 生产者
9  */
10 public class Task01 {
11     // 队列名称
12     public static final String QUEUE_NAME = "hello";
13
14     // 发送大量消息
15     public static void main(String[] args) throws Exception{
16         Channel channel = RabbitMQUtils.getChannel();
17
18         /**
19          * 生成一个队列
20          * 参数1: 队列名称
21          * 参数2: 队列里面的消息是否持久化(true磁盘), 默认情况消息存储在内存中(false)

```

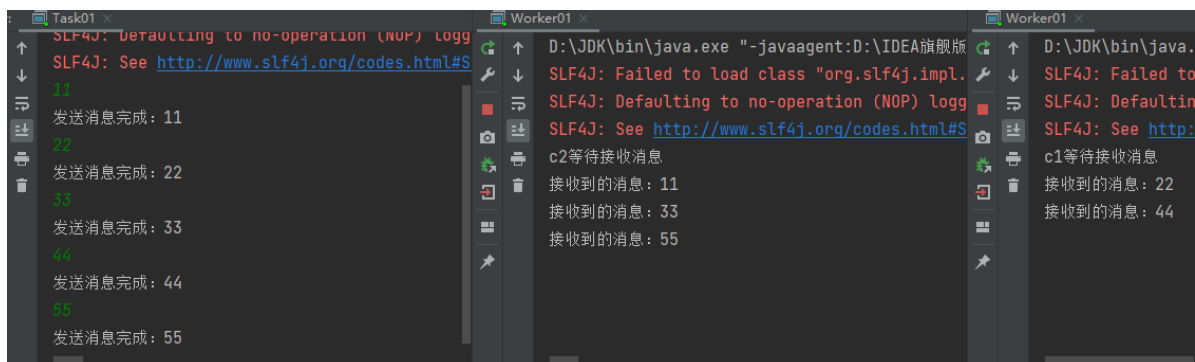
```

22      * 参数3: 该队列是否只供一个消费者进行消费(消息共享), true可以多个消费者消费,
false只能一个消费者消费
23      * 参数4: 是否自动删除, 最后一个消费者断开连接以后, 该队列是否自动删除, true是,
false不是
24      * 参数5: 其它参数
25      */
26      channel.queueDeclare(QUEUE_NAME, false, false, false, null);
27
28      // 从控制台接受消息
29      Scanner scanner = new Scanner(System.in);
30      while (scanner.hasNext()){
31          String message = scanner.next();
32          /**
33           * 发送一个消息
34           * 参数1: 送到哪个交换机
35           * 参数2: 路由的key值是哪个, 本次是队列名称
36           * 参数3: 其它的参数信息
37           * 参数4: 发送消息的消息体
38           */
39          channel.basicPublish("", QUEUE_NAME, null, message.getBytes());
40          System.out.println("发送消息完成: "+message);
41      }
42  }
43 }

```

运行结果

启动两个消费者线程



消息应答

消费者完成一个任务可能需要一段时间, 如果其中一个消费者处理一个长的任务并仅只完成了部分突然它挂掉了, 会发生什么情况。RabbitMQ 一旦向消费者传递

了一条消息, 便立即将该消息标记为删除。在这种情况下, 突然有个消费者挂掉了, 我们将丢失正在处理的消息。以及后续发送给该消费这的消息, 因为它无法接收到。

为了保证消息在发送过程中不丢失, 引入消息应答机制, 消息应答就是: **消费者在接收到消息并且处理该消息之后, 告诉 rabbitmq 它已经处理了, rabbitmq 可**

以把该消息删除了。

自动应答

消息发送后立即被认为已经传送成功，这种模式需要在**高吞吐量**和**数据传输安全性**方面做权衡,因为这种模式如果消息在接收到之前，消费者那边出现连接或者

channel 关闭，那么消息就丢失了,当然另一方面这种模式消费者那边可以传递过载的消息，**没有对传递的消息数量进行限制**，当然这样有可能使得消费者这边由

于接收太多还来不及处理的消息，导致这些消息的积压，最终使得内存耗尽，最终这些消费者线程被操作系统杀死，**所以这种模式仅适用在消费者可以高效并以**

某种速率能够处理这些消息的情况下使用。

消息应答的方法

1. Channel.basicAck(用于肯定确认)

RabbitMQ 已知道该消息并且成功的处理消息，可以将其丢弃了

2. Channel.basicNack(用于否定确认)


3. Channel.basicReject(用于否定确认)

与 Channel.basicNack 相比少一个参数 不处理该消息了直接拒绝，可以将其丢弃了

Multiple 的解释：

手动应答的好处是可以批量应答并且减少网络拥堵

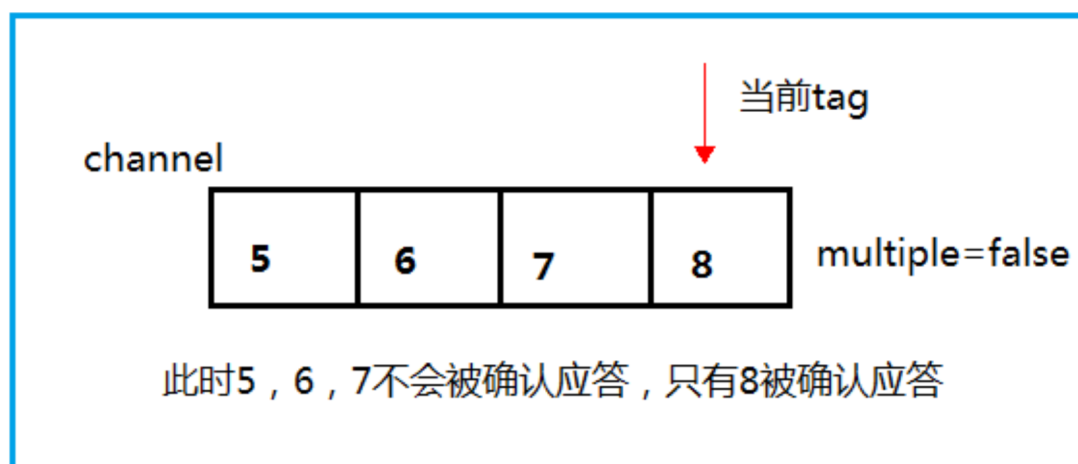
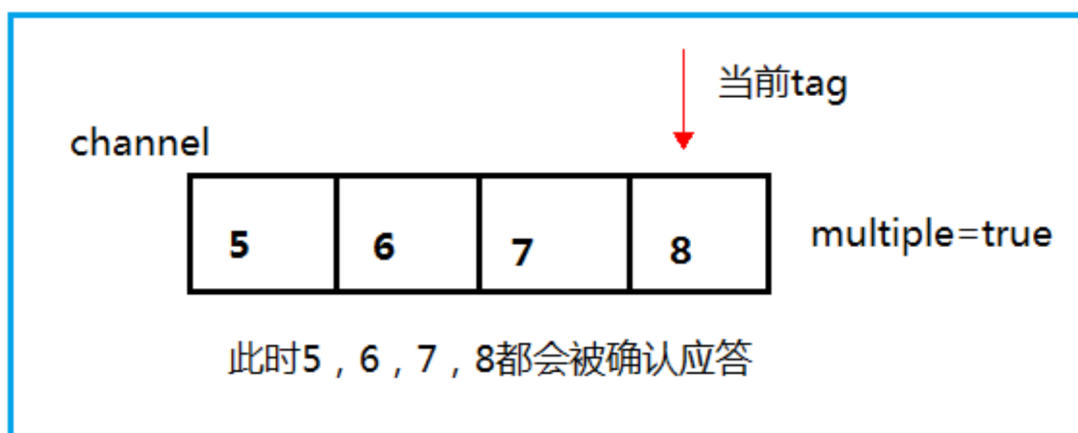
```
// positively acknowledge all deliveries up to
// this delivery tag
channel.basicAck(deliveryTag, true);
```



- true 代表批量应答 channel 上未应答的消息

比如说 channel 上有传送 tag 的消息 5,6,7,8 当前 tag 是8 那么此时5-8 的这些还未应答的消息都会被确认收到消息应答

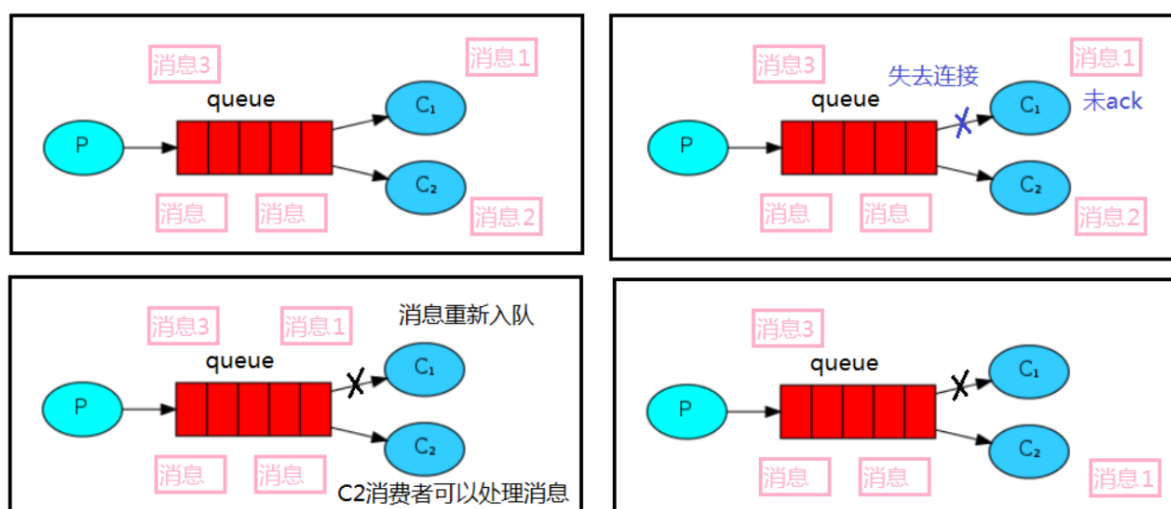
- false 同上面相比只会应答 tag=8 的消息 5,6,7 这三个消息依然不会被确认收到消息应答



消息自动重新入队

如果消费者由于某些原因失去连接(其通道已关闭, 连接已关闭或 TCP 连接丢失), 导致消息未发送 ACK 确认, RabbitMQ 将了解到消息未完全处理, 并将对其重新

排队。如果此时其他消费者可以处理, 它将很快将其重新分发给另一个消费者。这样, 即使某个消费者偶尔死亡, 也可以确保不会丢失任何消息。



消息手动应答代码

默认消息采用的是自动应答, 所以我们要想实现消息消费过程中不丢失, 需要把自动应答改为手动应答
消费者在上面代码的基础上增加了以下内容:

```
1 channel.basicAck(delivery.getEnvelope().getDeliveryTag(), false);
```

消息生产者

```
1 /**
2  * @Author: ww
3  * @DateTime: 2022/4/27 20:10
4  * @Description: 生产者
5  * @Target: 消息在手动应答时是不丢失的、返回队列中重新消费
6  */
7 public class Task2 {
8     // 队列名称
9     public static final String TASK_QUEUE_NAME = "ack_queue";
10    public static void main(String[] args) throws Exception{
11        Channel channel = RabbitMqUtils.getChannel();
12        // 声明队列
13        channel.queueDeclare(TASK_QUEUE_NAME, false, false, false, null);
14        // 从控制台输入信息
15        Scanner scanner = new Scanner(System.in);
16        while (scanner.hasNext()){
17            String message = scanner.next();
18            channel.basicPublish("", TASK_QUEUE_NAME,
19                null, message.getBytes("UTF-8"));
20            System.out.println("生产者发出消息: "+message);
21        }
22    }
```

消费者1

```
1 /**
2  * @Author: ww
3  * @DateTime: 2022/4/27 20:19
4  * @Description: 消费者
5  * @Target: 消息在手动应答时是不丢失的、返回队列中重新消费
6  */
7 public class work03 {
8     // 队列名称
9     public static final String TASK_QUEUE_NAME = "ack_queue";
10
11    // 接收消息
12    public static void main(String[] args) throws Exception{
13        Channel channel = RabbitMqUtils.getChannel();
14        System.out.println("C1等待接收消息处理时间较短");
15        DeliverCallback deliverCallback = (consumerTag, message)->{
16            // 沉睡1秒
17            SleepUtils.sleep(1);
18            System.out.println("接收到的消息: " + new
19                String(message.getBody(), "UTF-8"));
20            /**
21             * 手动应答
22             * 1. 消息的标记 tag
23             * 2. 是否批量应答
24             */
25            channel.basicAck(message.getEnvelope().getDeliveryTag(), false);
26        };
27    }
```

```

26         // 采用手动应答
27         boolean autoAck = false;
28         channel.basicConsume(TASK_QUEUE_NAME, autoAck, deliverCallback,
(consumerTag)->{
29             System.out.println(consumerTag+"消费者取消消费接口回调逻辑");
30         });
31     }
32 }

```

消费者2

```

1  /**
2   * @Author: ww
3   * @DateTime: 2022/4/27 20:19
4   * @Description: 消费者
5   * @Target: 消息在手动应答时是不丢失的、返回队列中重新消费
6   */
7  public class work04 {
8      // 队列名称
9      public static final String TASK_QUEUE_NAME = "ack_queue";
10
11     // 接收消息
12     public static void main(String[] args) throws Exception{
13         Channel channel = RabbitMQUtils.getChannel();
14         System.out.println("C2等待接收消息处理时间较长");
15
16         DeliverCallback deliverCallback = (consumerTag, message)->{
17             // 沉睡1秒
18             sleepUtils.sleep(20);
19             System.out.println("接收到的消息: " + new
String(message.getBody(), "UTF-8"));
20             /**
21              * 手动应答
22              * 1. 消息的标记 tag
23              * 2. 是否批量应答
24              */
25             channel.basicAck(message.getEnvelope().getDeliveryTag(), false);
26         };
27         // 采用手动应答
28         boolean autoAck = false;
29         channel.basicConsume(TASK_QUEUE_NAME, autoAck, deliverCallback,
(consumerTag)->{
30             System.out.println(consumerTag+"消费者取消消费接口回调逻辑");
31         });
32     }
33 }

```

睡眠工具类

```

1 public class SleepUtils {
2     public static void sleep(int second){
3         try {
4             Thread.sleep(1000*second);
5         } catch (InterruptedException _ignored) {
6             Thread.currentThread().interrupt();
7         }
8     }
9 }

```

应答效果

正常情况下消息发送方发送两个消息 C1 和 C2 分别接收到消息并进行处理

aa
生产者发出消息aa
bb
生产者发出消息bb

C1等待接收消息 处理时间较短.....
接收到消息aa

C2等待接收消息 处理时间较长.....
接收到消息bb

在发送者发送消息 dd，发出消息之后的把 C2 消费者停掉，按理说该 C2 来处理该消息，但是由于它处理时间较长，在还未处理完，也就是说 C2 还没有执行 ack

代码的时候，C2 被停掉了，此时会看到消息被 C1 接收到了，说明消息 dd 被重新入队，然后分配给能处理消息的 C1 处理了

Overview				Messages			Message rates				+/-
Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack		
task_queue	classic		idle	0	0	0					

Overview				Messages			Message rates			
Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack	
task_queue	classic		idle	0	1	1	0.00/s	0.00/s	0.00/s	

aa
生产者发出消息aa---> C1
bb
生产者发出消息bb---> C2
cc
生产者发出消息cc---> C1
dd
生产者发出消息dd---> C2

C1等待接收消息 处理时间较短.....
接收到消息aa
接收到消息cc
接收到消息dd 这里是C1在处理

C2等待接收消息 处理时间较长.....
接收到消息bb

队列持久化

当 RabbitMQ 服务停掉以后，消息生产者发送过来的消息不丢失要如何保障？默认情况下 RabbitMQ 退出或由于某种原因崩溃时，它忽视队列和消息，除非告知

它不要这样做。确保消息不会丢失需要做两件事：**我们需要将队列和消息都标记为持久化。**

队列如何实现持久化？

之前我们创建的队列都是非持久化的，rabbitmq 如果重启的话，该队列就会被删除掉，如果要队列实现持久化需要在声明队列的时候把 durable 参数设置为持久

化

```
1 // 队列持久化
2 boolean durable = true;
3 channel.queueDeclare(TASK_QUEUE_NAME, durable, false, false, null);
```

注意：如果之前声明的队列不是持久化的，需要把原先队列先删除，或者重新创建一个持久化的队列，不然就会出现错误

ITON_FAILED - inequivalent arg 'durable' for queue 'task_queue' in vhost '/': received 'true' but current is 'false'.

以下为控制台中持久化与非持久化队列的 UI 显示区：

Overview				Messages			Message rates		
Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
task_queue	classic		idle	0	0	0	0.00/s	0.00/s	0.00/s

Overview				Messages			Message rates		
Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
task_queue	classic	D	idle	0	0	0			

这个时候即使重启 rabbitmq 队列也依然存在。

消息持久化

要想让消息实现持久化需要在消息生产者修改代码，MessageProperties.PERSISTENT_TEXT_PLAIN 添加这个属性。

```
1 // 前提，实现队列持久化
2 channel.basicPublish("", TASK_QUEUE_NAME, null, message.getBytes("UTF-8"));
3 // 修改后
4 channel.basicPublish("", TASK_QUEUE_NAME,
    MessageProperties.PERSISTENT_TEXT_PLAIN, message.getBytes("UTF-8"));
```

将消息标记为持久化并不能完全保证不会丢失消息。尽管它告诉 RabbitMQ 将消息保存到磁盘，但是这里依然存在当消息刚准备存储在磁盘的时候 但是还没有存

储完，消息还在缓存的一个间隔点。此时并没有真正写入磁盘。持久性保证并不强，但是对于我们的简单任务队列而言，这已经绰绰有余了。后续有更好的策

略。

不公平分发

在最开始的时候我们学习到 RabbitMQ 分发消息采用的轮训分发，但是在某种场景下这种策略并不是很好，比方说有两个消费者在处理任务，其中有个消费者 1

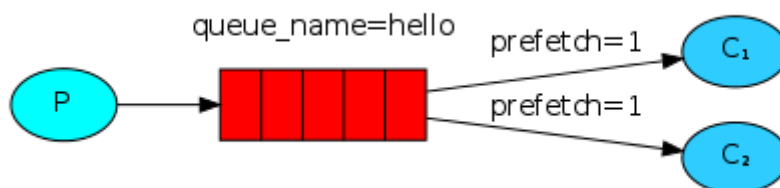
处理任务的速度非常快，而另外一个消费者 2 处理速度却很慢，这个时候我们还是采用轮训分发的话就会到这处理速度快的这个消费者很大一部分时间处于空闲

状态，而处理慢的那个消费者一直在干活，这种分配方式在这种情况下其实就不太好，但是 RabbitMQ 并不知道这种情况它依然很公平的进行分发。

为了避免这种情况，我们可以设置参数 `channel.basicQos(1)`

```
1 //不公平分发
2 int prefetchCount = 1;
3 channel.basicQos(prefetchCount);
```

Consumers							
Channel	Consumer tag	Ack required	Exclusive	Prefetch count	Active ?	Activity status	Arguments
		•	○	1	•	up	
		•	○	1	•	up	



意思就是如果这个任务我还没有处理完或者我还没有应答你，你先别分配给我，我目前只能处理一个任务，然后 rabbitmq 就会把该任务分配给没有那么忙的那个

空闲消费者，当然如果所有的消费者都没有完成手上任务，队列还在不停的添加新任务，队列有可能就会遇到队列被撑满的情况，这个时候就只能添加新的

worker 或者改变其他存储任务的策略。

预取值分发

带权的消息分发

本身消息的发送就是异步发送的，所以在任何时候，channel 上肯定不止只有一个消息另外来自消费者的手动确认本质上也是异步的。因此这里就存在一个未确认

的消息缓冲区，因此希望开发人员能**限制此缓冲区的大小，以避免缓冲区里面无限制的未确认消息问题**。这个时候就可以通过使用 `basic.qos` 方法设置“预取计

数”值来完成的。

该值定义通道上允许的未确认消息的最大数量。一旦数量达到配置的数量，RabbitMQ 将停止在通道上传递更多消息，除非至少有一个未处理的消息被确认，例

如，假设在通道上有未确认的消息 5、6、7、8，并且通道的预取计数设置为 4，此时 RabbitMQ 将不会在该通道上再传递任何消息，除非至少有一个未应答的消

息被 ack。比方说 tag=6 这个消息刚刚被确认 ACK，RabbitMQ 将会感知 这个情况到并再发送一条消息。消息应答和 QoS 预取值对用户吞吐量有重大影响。

通常，增加预取将提高 向消费者传递消息的速度。**虽然自动应答传输消息速率是最佳的，但是，在这种情况下已传递但尚未处理的消息的数量也会增加，从而增**

加了消费者的 RAM 消耗(随机存取存储器)应该小心使用具有无限预处理的自动确认模式或手动确认模式，消费者消费了大量的消息如果没有确认的话，会导致消

费者连接节点的 内存消耗变大，所以找到合适的预取值是一个反复试验的过程，不同的负载该值取值也不同 100 到 300 范 围内的值通常可提供最佳的吞吐量，并

且不会给消费者带来太大的风险。

预取值为 1 是最保守的。当然这将使吞吐量变得很低，特别是消费者连接延迟很严重的情况下，特别是在消费者连接等待时间较长的环境 中。对于大多数应用来

说，稍微高一点的值将是最佳的。



RabbitMQ-发布确认

发布确认原理

生产者将信道设置成 confirm 模式，一旦信道进入 confirm 模式，所有在该信道上发布消息都将会被指派一个唯一的 ID(从 1 开始)，一旦消息被投递到所有匹

配的队列之后，broker 就会发送一个确认给生产者(包含消息的唯一 ID)，这就使得生产者知道消息已经正确到达目的队列了，如果消息和队列是可持久化的，那么

确认消息会在将消息写入磁盘之后发出，broker 回传给生产者的确认消息中 delivery-tag 域包含了确认消息的序列号，此外 broker 也可以设置basic.ack 的

multiple 域，表示到这个序列号之前的所有消息都已经得到了处理。

confirm 模式最大的好处在于他是异步的，一旦发布一条消息，生产者应用程序就可以在等信道返回确认的同时继续发送下一条消息，当消息最终得到确认之后，

生产者应用便可以通过回调方法来处理该确认消息，如果RabbitMQ 因为自身内部错误导致消息丢失，就会发送一条 nack 消息，生产者应用程序同样可以在回调

方法中处理该 nack 消息。

发布确认的策略

开启发布确认的方法

发布确认默认是没有开启的，如果要开启需要调用方法 `confirmSelect`，每当你要想使用发布 确认，都需要在 `channel` 上调用该方法

```
1 // 开启发布确认
2 channel.confirmSelect();
```

单个确认发布

这是一种简单的确认方式，它是一种**同步**确认发布的方式，也就是发布一个消息之后只有它 被确认发布，后续的消息才能继续发布，`waitForConfirmsOrDie(long)`

这个方法只有在消息被确认 的时候才返回，如果在指定时间范围内这个消息没有被确认那么它将抛出异常。

这种确认方式有一个最大的缺点就是：**发布速度特别的慢**，因为如果没有确认发布的消息就会 阻塞所有后续消息的发布，这种方式最多提供每秒不超过数百条发布

消息的吞吐量。当然对于某些应用程序来说这可能已经足够了。

```
1 // 单个发布确认
2 public static void publishMessageIndividually() throws Exception{
3     Channel channel = RabbitMqutils.getChannel();
4     // 队列声明
5     String queueName = UUID.randomUUID().toString();
6     channel.queueDeclare(queueName, true, false, false, null);
7     // 开启发布确认
8     channel.confirmSelect();
9     // 开始时间
10    long begin = System.currentTimeMillis();
11    // 批量发消息 单个发布确认
12    for (int i = 0; i < MESSAGE_COUNT; i++) {
13        String message = i + "";
14        channel.basicPublish("", queueName, null, message.getBytes());
15        // 单个消息 就马上进行发布确认
16        boolean flag = channel.waitForConfirms();
17        if (flag){
18            System.out.println("消息发布成功");
19        }
20    }
21    // 结束时间
22    long end = System.currentTimeMillis();
23    System.out.println("发布" + MESSAGE_COUNT + "条单独确认消息，耗时：" +
24        (end - begin) + "s");
25 }
```

批量确认发布

上面那种方式非常慢，与单个等待确认消息相比，先发布一批消息然后一起确认可以极大地 提高吞吐量，当然这种方式的缺点就是：当发生故障导致发布出现问题

时，不知道是哪个消息出现 问题了，我们必须将整个批处理保存在内存中，以记录重要的信息而后重新发布消息。当然这种 方案仍然是同步的，也一样阻塞消息

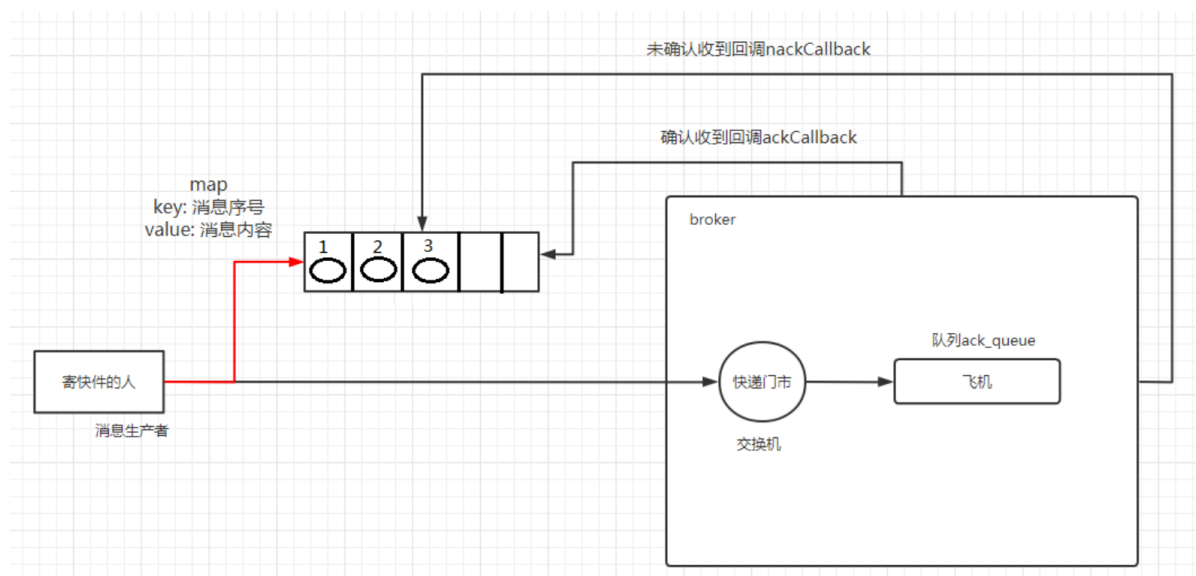
的发布。

```
1 // 批量发布确认
2 public static void publishMessageBatch() throws Exception {
3     Channel channel = RabbitMqUtils.getChannel();
4     // 队列声明
5     String queueName = UUID.randomUUID().toString();
6     channel.queueDeclare(queueName, true, false, false, null);
7     // 开启发布确认
8     channel.confirmSelect();
9     // 开始时间
10    long begin = System.currentTimeMillis();
11    // 批量确认消息大小
12    int batchSize = 50;
13    // 批量发消息 批量发布确认
14    for (int i = 0; i < MESSAGE_COUNT; i++) {
15        String message = i + "";
16        channel.basicPublish("", queueName, null, message.getBytes());
17        // 判断达到50条消息的时候 批量确认一次
18        if (i % batchSize == 0) {
19            channel.confirmSelect();
20        }
21    }
22    // 结束时间
23    long end = System.currentTimeMillis();
24    System.out.println("发布" + MESSAGE_COUNT + "条批量确认消息, 耗时: " +
25        (end - begin) + "s");
26 }
```

异步确认发布

异步确认虽然编程逻辑比上两个要复杂，但是性价比最高，无论是可靠性还是效率都没得说，他是利用回调函数来达到消息可靠性传递的，这个中间件也是通过

函数回调来保证是否投递成功。



```
1 // 异步发布确认
2 public static void publishMessageAsync() throws Exception {
3     Channel channel = RabbitMqUtils.getChannel();
4     // 队列声明
```

```

5      String queueName = UUID.randomUUID().toString();
6      channel.queueDeclare(queueName, true, false, false, null);
7      // 开启发布确认
8      channel.confirmSelect();
9      /**
10     * 线程安全有序的一个哈希表 map 适用于高并发的情况
11     * 功能1: 使序号和消息进行关联
12     * 功能2: 批量删除条目
13     * 功能3: 支持高并发(多线程)
14     */
15     ConcurrentSkipListMap<Long, String> outstandingConfirms = new
ConcurrentSkipListMap<>();
16     // 消息确认成功 回调函数
17     ConfirmCallback ackCallback = (deliveryTag, multiple) -> {
18         // 如果是批量 就批量删除消息
19         if (multiple) {
20             // 第二步: 删除已经确认的消息
21             ConcurrentNavigableMap<Long, String> confirmed =
outstandingConfirms.headMap(deliveryTag);
22             confirmed.clear();
23         } else {
24             // 单个
25             outstandingConfirms.remove(deliveryTag);
26         }
27         System.out.println("确认的消息: " + deliveryTag);
28     };
29     // 消息确认失败 回调函数
30     /**
31     * 参数1: 消息的标记
32     * 参数2: 是否为批量确认
33     */
34     ConfirmCallback nackCallback = (deliveryTag, multiple) -> {
35         // 第三步: 打印未确认的消息
36         String message = outstandingConfirms.get(deliveryTag);
37         System.out.println("未确认的消息: " + message + "未确认的标记: " +
deliveryTag);
38     };
39
40     // 准备消息的监听器 监听哪些消息成功了 异步通知
41     /**
42     * 参数1: 监听哪些消息成功了
43     * 参数2: 监听哪些消息失败了
44     */
45     channel.addConfirmListener(ackCallback, nackCallback);
46     // 开始时间
47     long begin = System.currentTimeMillis();
48     // 批量发消息 异步
49     for (int i = 0; i < MESSAGE_COUNT; i++) {
50         String message = "消息" + i;
51         // 第一步: 此处记录下所有要发送的消息
52         outstandingConfirms.put(channel.getNextPublishSeqNo(), message);
53
54         channel.basicPublish("", queueName, null, message.getBytes());
55     }
56     // 结束时间
57     long end = System.currentTimeMillis();
58     System.out.println("发布" + MESSAGE_COUNT + "条异步确认消息, 耗时: " +
(end - begin) + "s");

```

如何处理异步未确认消息

最好的解决的解决方案就是把未确认的消息放到一个基于内存的能被发布线程访问的队列，比如说用 `ConcurrentLinkedQueue` 这个队列在 `confirm callbacks` 与

发布线程之间进行消息的传递。

操作步骤：

1. 记录下所有要发送的消息
2. 删除已经确认的消息
3. 打印未确认的消息(可有可无)

三种发布确认策略速度对比

- 单独发布消息
同步等待确认，简单，但吞吐量非常有限。
- 批量发布消息
批量同步等待确认，简单，合理的吞吐量，一旦出现问题但很难推断出是那条消息出现了问题。
- 异步处理
最佳性能和资源使用，在出现错误的情况下可以很好地控制，但是实现起来稍微难些

```
1 // 1. 单个确认 发布200条单独确认消息，耗时：3978s
2 ConfirmMessage.publishMessageIndividually();
3 // 2. 批量确认 发布200条批量确认消息，耗时：106s
4 ConfirmMessage.publishMessageBatch();
5 // 3. 异步批量确认 发布200条异步确认消息，耗时：10s
6 ConfirmMessage.publishMessageAsync();
```

RabbitMQ-交换机

在以上的案例中，我们创建了一个工作队列。我们假设的是工作队列背后，每个任务都恰好交付给一个消费者(工作进程)。在这一部分中，我们将做一些完全不同的

的事情-我们将消息传达给多个消费者。这种模式 称为“发布/订阅”。

为了说明这种模式，我们将构建一个简单的日志系统。它将由两个程序组成:第一个程序将发出日志消息，第二个程序是消费者。其中我们会启动两个消费者，其

中一个消费者接收到消息后把日志存储在磁盘，另外一个消费者接收到消息后把消息打印在屏幕上，事实上第一个程序发出的日志消息将广播给所有消费者。

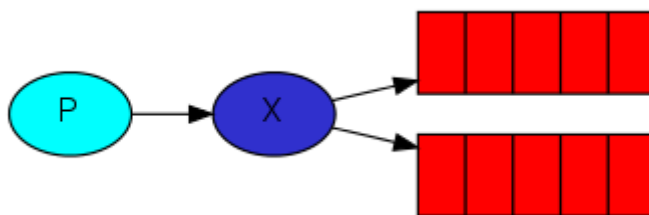
Exchanges

概念

RabbitMQ 消息传递模型的核心思想是: **生产者生产的消息从不会直接发送到队列**。实际上, 通常生产者甚至都不知道这些消息传递到了哪些队列中。

相反, **生产者只能将消息发送到交换机(exchange)**, 交换机工作的内容非常简单, 一方面它接收来自生产者的消息, 另一方面将它们推入队列。交换机必须确切

知道如何处理收到的消息。是应该把这些消息放到特定队列还是说把他们到许多队列中还是说应该丢弃它们。这就要由交换机的类型来决定。



Exchanges的类型

- 直接(direct)
- 主题(topic)
- 标题(headers)
- 扇出(fanout)

无名Exchange

之前能实现的原因是因为我们使用的是默认交换, 我们通过空字符串("")进行标识。

```
1 | channel.basicPublish("", queueName, null, message.getBytes());
```

第一个参数是交换机的名称。空字符串表示默认或无名称交换机: 消息能路由发送到队列中其实是由 routingKey(bindingkey)绑定 key 指定的, 如果它存在的话

临时队列

之前的章节我们使用的是具有特定名称的队列(hello 和 ack_queue)。队列的名称我们来说至关重要, 我们需要指定我们的消费者去消费哪个队列的消息。

每当我们连接到 Rabbit 时, 我们都需要一个全新的空队列, 为此我们可以创建一个具有**随机名称的队列**, 或者能让服务器为我们选择一个随机队列名称那就更好

了。其次**一旦我们断开了消费者的连接, 队列将被自动删除**。

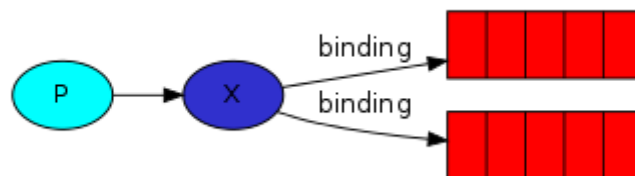
创建临时队列的方式如下:

```
1 | String queueName = channel.queueDeclare().getQueue();
```

绑定(bindings)

binding 其实是 exchange 和 queue 之间的桥梁，它告诉我们 exchange 和那个队 列进行了绑定关系。比如说下面这张图告诉我们的就是 X 与 Q1 和 Q2 进行了

绑定



Fanout exchange

介绍

Fanout 这种类型非常简单。正如从名称中猜到的那样，它是将接收到的所有消息广播到它知道的 所有队列中。系统中默认有些 exchange 类型

Exchanges

▼ All exchanges (10)

Pagination

Page 1 of 1 - Filter: ☐ Regex ?

Name	Type	Features	Message rate in	Message rate out	+/-
(AMQP default)	direct	D	0.00/s	0.00/s	
amq.direct	direct	D			
amq.fanout	fanout	D			
amq.headers	headers	D			
amq.match	headers	D			
amq.rabbitmq.trace	topic	D I			
amq.topic	topic	D			
direct_logs	direct		0.00/s	0.00/s	
logs	fanout		0.00/s	0.00/s	
topic_logs	topic		0.00/s	0.00/s	

实现



Logs 和临时队列的绑定关系如下图：

▼ Bindings

This exchange

⇓

To	Routing key	Arguments	
amq.gen-EDKbiyHhOXSONkDd5iGtuA			Unbind
amq.gen-rMe4lrI64xJ3ZP-QBn8AlA			Unbind

ReceiveLogs01

```

1  /**
2   * @Author: ww
3   * @DateTime: 2022/4/30 14:39
4   * @Description: 消息接收
5   */
6  public class ReceiveLogs01 {
7      //交换机名称
8      public static final String EXCHANGE_NAME = "logs";
9
10     public static void main(String[] args) throws Exception{
11         Channel channel = RabbitMqutils.getChannel();
12         // 声明一个交换机
13         channel.exchangeDeclare(EXCHANGE_NAME, "fanout");
14         /**
15          * 声明一个队列 临时队列
16          * 队列的名称是随机的
17          * 当消费者断开与队列的连接的时候，队列就自动删除
18          */
19         String queueName = channel.queueDeclare().getQueue();
20
21         /**
22          * 绑定交换机与队列
23          */
24         channel.queueBind(queueName, EXCHANGE_NAME, "");
25         System.out.println("c1等待接收消息，打印接收到的消息....");
26
27         // 声明 接收消息
28         DeliverCallback deliverCallback = (consumerTag, message)->{
29             System.out.println("c1已接收消息: " + new
30             String(message.getBody(), "UTF-8"));
31         };
32     }
33 }

```

```

31         // 消费者取消消息
32
33         // 接收消息
34         channel.basicConsume(queueName, true, deliverCallback, consumerTag ->
35             {});
36     }

```

ReceiveLogs02

```

1  /**
2   * @Author: ww
3   * @DateTime: 2022/4/30 14:39
4   * @Description: 消息接收
5   */
6  public class ReceiveLogs02 {
7      //交换机名称
8      public static final String EXCHANGE_NAME = "logs";
9
10     public static void main(String[] args) throws Exception{
11         Channel channel = RabbitMqUtils.getChannel();
12         // 声明一个交换机
13         channel.exchangeDeclare(EXCHANGE_NAME, "fanout");
14         /**
15          * 声明一个队列 临时队列
16          * 队列的名称是随机的
17          * 当消费者断开与队列的连接的时候，队列就自动删除
18          */
19         String queueName = channel.queueDeclare().getQueue();
20
21         /**
22          * 绑定交换机与队列
23          */
24         channel.queueBind(queueName, EXCHANGE_NAME, "");
25         System.out.println("c2等待接收消息，打印接收到的消息....");
26
27         // 声明 接收消息
28         DeliverCallback deliverCallback = (consumerTag, message)->{
29             System.out.println("c2已接收消息: " + new
30                 String(message.getBody(), "UTF-8"));
31         };
32         // 消费者取消消息
33
34         // 接收消息
35         channel.basicConsume(queueName, true, deliverCallback, consumerTag ->
36             {});
37     }

```

EmitLog

```

1  /**
2   * @Author: ww
3   * @DateTime: 2022/4/30 15:02
4   * @Description: 发消息给交换机
5   */

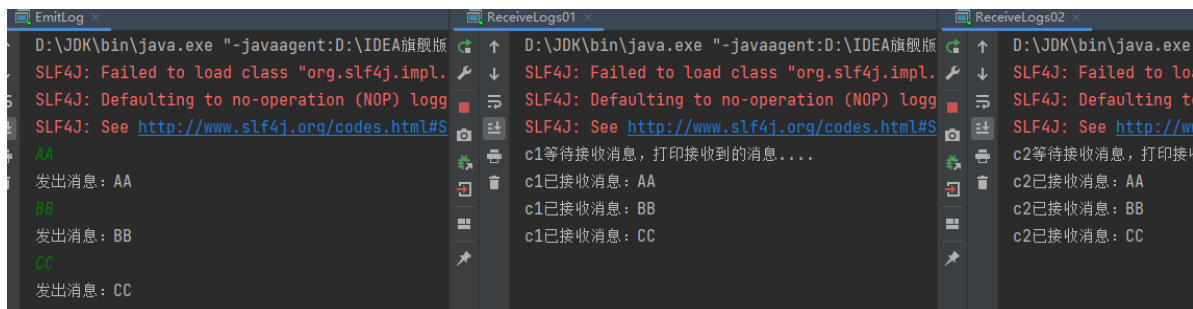
```

```

6 public class EmitLog {
7     //交换机名称
8     public static final String EXCHANGE_NAME = "logs";
9
10    public static void main(String[] args) throws Exception{
11        channel channel = RabbitMQUtils.getChannel();
12        channel.exchangeDeclare(EXCHANGE_NAME,"fanout");
13
14        Scanner scanner = new Scanner(System.in);
15
16        while (scanner.hasNext()){
17            String message = scanner.next();
18
19            channel.basicPublish(EXCHANGE_NAME,"",null,message.getBytes("UTF-8"));
20            System.out.println("发出消息: "+ message);
21        }
22    }
23 }

```

结果:



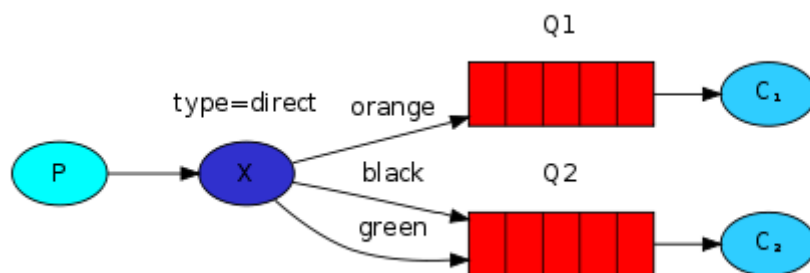
Direct exchange

介绍

上一节中的我们的日志系统将所有消息广播给所有消费者，对此我们想做一些改变，例如我们希望将日志消息写入磁盘的程序仅接收严重错误(errors)，而不存储

哪些警告(warning)或信息(info)日志 消息避免浪费磁盘空间。Fanout 这种交换类型并不能给我们带来很大的灵活性-它只能进行无意识的广播，在这里我们将使用

direct 这种类型来进行替换，这种类型的工作方式是，消息只去到它绑定的 routingKey 队列中去。

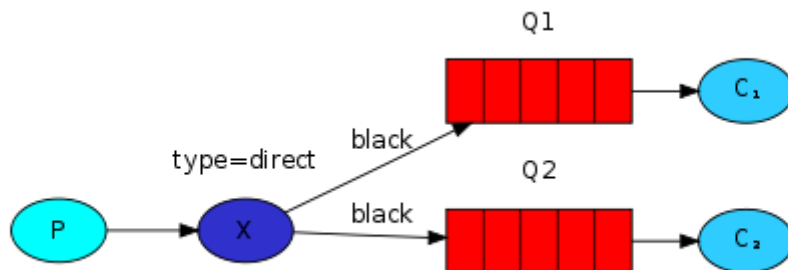


在上面这张图中，我们可以看到 X 绑定了两个队列，绑定类型是 direct。队列 Q1 绑定键为 orange，队列 Q2 绑定键有两个:一个绑定键为 black，另一个绑定键

为 green。

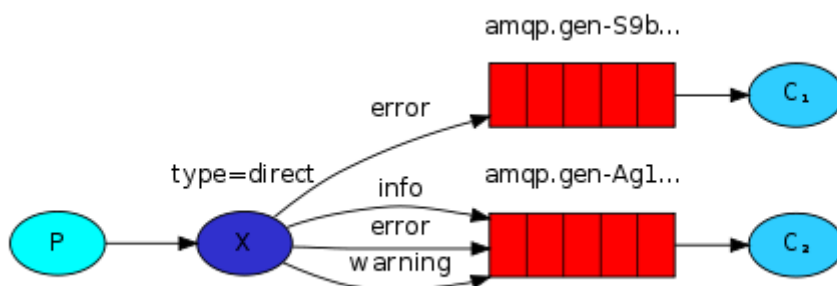
在这种绑定情况下，生产者发布消息到 exchange 上，绑定键为 orange 的消息会被发布到队列 Q1。绑定键为 blackgreen 和的消息会被发布到队列 Q2，其他消息类型的消息将被丢弃。

多重绑定



当然如果 exchange 的绑定类型是 direct，但是它绑定的多个队列的 key 如果都相同，在这种情况下虽然绑定类型是 direct 但是它表现的就和 fanout 有点类似了，就跟广播差不多。

实现



Exchange: direct_logs

This exchange			
⇓			
To	Routing key	Arguments	
console	info		Unbind
console	warning		Unbind
disk	error		Unbind

c1: 绑定console, routingKey为info、warning

c2: 绑定disk, routingKey为error

ReceiveLogsDirect01

```
1 public class ReceiveLogsDirect01 {
2     public static final String EXCHANGE_NAME = "direct_logs";
3
4     public static void main(String[] args) throws Exception{
5         Channel channel = RabbitMQUtils.getChannel();
6         // 声明一个交换机
7         channel.exchangeDeclare(EXCHANGE_NAME, BuiltinExchangeType.DIRECT);
8         // 声明一个队列
9         channel.queueDeclare("console", false, false, false, null);
10        channel.queueBind("console", EXCHANGE_NAME, "info");
11        channel.queueBind("console", EXCHANGE_NAME, "warning");
12
13        // 声明 接收消息
14        DeliverCallback deliverCallback = (consumerTag, message)->{
15            System.out.println("c1已接收消息: " + new
String(message.getBody(), "UTF-8"));
16        } ;
17        // 接收消息
18        channel.basicConsume("console", true, deliverCallback, consumerTag ->
{});
19    }
20 }
```

ReceiveLogsDirect02

```
1 public class ReceiveLogsDirect02 {
2     public static final String EXCHANGE_NAME = "direct_logs";
3
4     public static void main(String[] args) throws Exception{
5         Channel channel = RabbitMQUtils.getChannel();
6         // 声明一个交换机
7         channel.exchangeDeclare(EXCHANGE_NAME, BuiltinExchangeType.DIRECT);
8         // 声明一个队列
9         channel.queueDeclare("disk", false, false, false, null);
10        channel.queueBind("disk", EXCHANGE_NAME, "error");
11
12        // 声明 接收消息
13        DeliverCallback deliverCallback = (consumerTag, message)->{
14            System.out.println("c2已接收消息: " + new
String(message.getBody(), "UTF-8"));
15        } ;
16        // 接收消息
17        channel.basicConsume("disk", true, deliverCallback, consumerTag -> {});
18    }
19 }
```

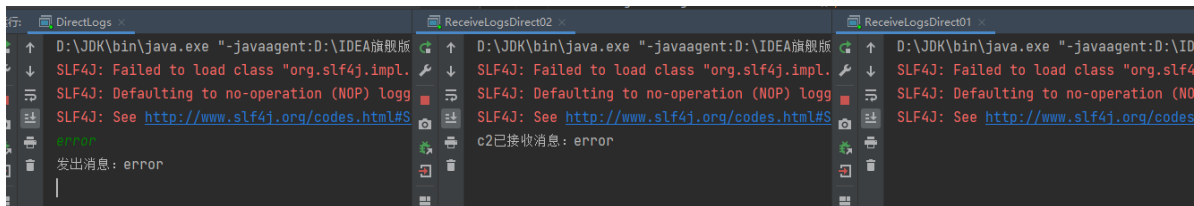
EmitLogDirect

```

1 public class DirectLogs {
2     //交换机名称
3     public static final String EXCHANGE_NAME = "direct_logs";
4     public static void main(String[] args) throws Exception{
5         Channel channel = RabbitMQUtils.getChannel();
6         Scanner scanner = new Scanner(System.in);
7         while (scanner.hasNext()){
8             String message = scanner.next();
9
10            channel.basicPublish(EXCHANGE_NAME,"error",null,message.getBytes("UTF-8"));
11            System.out.println("发出消息: "+ message);
12        }
13    }
14 }

```

结果



Topics exchange

介绍

在上一个一节中，我们改进了日志记录系统。我们没有使用只能进行随意广播的 fanout 交换机，而是使用了 direct 交换机，从而有能实现有选择性地接收日志。

尽管使用 direct 交换机改进了我们的系统，但是它仍然存在局限性——比方说我们想接收的日志类型有 info.base 和 info.advantage，某个队列只想 info.base 的

消息，那这个时候direct 就办不到了。这个时候就只能使用 **topic** 类型。

Topic的要求

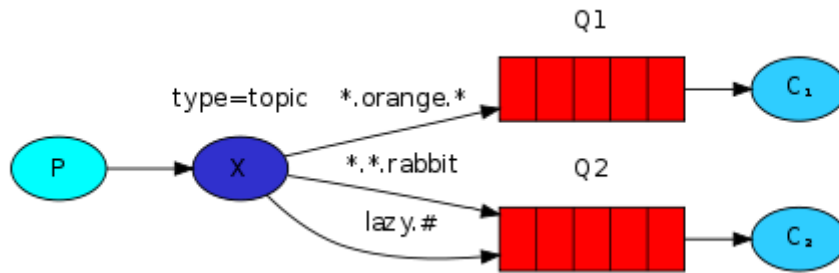
发送到类型是 topic 交换机的消息的 routing_key 不能随意写，必须满足一定的要求，它必须是一个**单词列表，以点号分隔开**。这些单词可以是任意单词

比如说: "stock.usd.nyse", "nyse.vmw", "quick.orange.rabbit".这种类型的。

当然这个单词列表最多不能超过 255 个字节。

在这个规则列表中，其中有两个替换符是大家需要注意的：

- *(星号)可以代替一个单词
- #(井号)可以替代零个或多个单词



在这个例子中，我们将发送所有描述动物的消息。消息将使用由三个单词（两个点）组成的路由键发送。路由键中的第一个词将描述速度，第二个是颜色，第三个是物种：“..”。

我们创建了三个绑定：Q1 与绑定键“`.orange.*`”绑定，Q2 与“`..rabbit`”和“`lazy.#`”绑定。

这些绑定可以概括为：

- Q1 对所有橙色动物都感兴趣。
- Q2 想听听关于兔子的一切，以及关于懒惰动物的一切。

注意：

- 当一个队列绑定键是#，那么这个队列将接收所有数据，就有点像 fanout 了
- 如果队列绑定键当中没有#和*出现，那么该队列绑定类型就是 direct 了

实现

Exchange: topic_logs

This exchange			
⇕			
To	Routing key	Arguments	
Q1	*.orange.*		Unbind
Q2	*.*.rabbit		Unbind
Q2	lazy.#		Unbind

ReceiveLogsTopic01

```

1  /**
2   * @Author: ww
3   * @DateTime: 2022/4/30 15:38
4   * @Description: 声明主体交换机及相关队列 消费者c1
5   */
6  public class ReceiveLogsTopic01 {
7      public static final String EXCHANGE_NAME = "topic_logs";
8
9      public static void main(String[] args) throws Exception{

```

```

10     Channel channel = RabbitMqUtils.getChannel();
11     // 声明交换机
12     channel.exchangeDeclare(EXCHANGE_NAME, BuiltinExchangeType.TOPIC);
13     // 声明队列
14     String queueName = "q1";
15     channel.queueDeclare(queueName, false, false, false, null);
16     channel.queueBind(queueName, EXCHANGE_NAME, "*.orange.*");
17     System.out.println("c1等待接收消息....");
18
19     // 声明接收消息
20     DeliverCallback deliverCallback = (consumerTag, message) ->{
21         System.out.println(new String(message.getBody(), "utf-8"));
22         System.out.println("接收队列: " + queueName + ", 绑定键: " +
message.getEnvelope().getRoutingKey());
23     };
24
25     // 接收消息
26     channel.basicConsume(queueName, true, deliverCallback, consumerTag ->
{});
27 }
28 }

```

ReceiveLogsTopic02

```

1  /**
2   * @Author: ww
3   * @DateTime: 2022/4/30 15:38
4   * @Description: 声明主体交换机及相关队列 消费者c1
5   */
6  public class ReceiveLogsTopic02 {
7      public static final String EXCHANGE_NAME = "topic_logs";
8
9      public static void main(String[] args) throws Exception{
10         Channel channel = RabbitMqUtils.getChannel();
11         // 声明交换机
12         channel.exchangeDeclare(EXCHANGE_NAME, BuiltinExchangeType.TOPIC);
13         // 声明队列
14         String queueName = "q2";
15         channel.queueDeclare(queueName, false, false, false, null);
16         channel.queueBind(queueName, EXCHANGE_NAME, ".*.rabbit");
17         channel.queueBind(queueName, EXCHANGE_NAME, "lazy.#");
18         System.out.println("c2等待接收消息....");
19
20         // 声明接收消息
21         DeliverCallback deliverCallback = (consumerTag, message) ->{
22             System.out.println(new String(message.getBody(), "utf-8"));
23             System.out.println("接收队列: " + queueName + ", 绑定键: " +
message.getEnvelope().getRoutingKey());
24         };
25
26         // 接收消息
27         channel.basicConsume(queueName, true, deliverCallback, consumerTag ->
{});
28     }
29 }

```

EmitLogTopic

```

1  /**
2   * @Author: ww
3   * @DateTime: 2022/4/30 15:49
4   * @Description: 生产者
5   */
6  public class EmitLogTopic {
7      public static final String EXCHANGE_NAME = "topic_logs";
8
9      public static void main(String[] args) throws Exception{
10         Channel channel = RabbitMQUtils.getChannel();
11         /**
12          * Q1-->绑定的是
13          * 中间带 orange 带 3 个单词的字符串(*.orange.*)
14          * Q2-->绑定的是
15          * 最后一个单词是 rabbit 的 3 个单词(*.*.rabbit)
16          * 第一个单词是 lazy 的多个单词(lazy.#)
17          */
18         Map<String, String> bindingKeyMap = new HashMap<>();
19         bindingKeyMap.put("quick.orange.rabbit","被队列 Q1Q2 接收到");
20         bindingKeyMap.put("lazy.orange.elephant","被队列 Q1Q2 接收到");
21         bindingKeyMap.put("quick.orange.fox","被队列 Q1 接收到");
22         bindingKeyMap.put("lazy.brown.fox","被队列 Q2 接收到");
23         bindingKeyMap.put("lazy.pink.rabbit","虽然满足两个绑定但只被队列 Q2 接收
一次");
24         bindingKeyMap.put("quick.brown.fox","不匹配任何绑定不会被任何队列接收到会
被丢弃");
25         bindingKeyMap.put("quick.orange.male.rabbit","是四个单词不匹配任何绑定会
被丢弃");
26         bindingKeyMap.put("lazy.orange.male.rabbit","是四个单词但匹配 Q2");
27         for (Map.Entry<String, String> bindingKeyEntry:
bindingKeyMap.entrySet()){
28             String bindingKey = bindingKeyEntry.getKey();
29             String message = bindingKeyEntry.getValue();
30             channel.basicPublish(EXCHANGE_NAME,bindingKey, null,
message.getBytes("UTF-8"));
31             System.out.println("生产者发出消息: " + message);
32         }
33     }
34 }

```

RabbitMQ-死信队列

死信的概念

先从概念解释上搞清楚这个定义，死信，顾名思义就是无法被消费的消息，字面意思可以这样理解，一般来说，producer 将消息投递到 broker 或者直接到

queue 里了，consumer 从 queue 取出消息 进行消费，但某些时候由于特定的原因**导致 queue 中的某些消息无法被消费**，这样的消息如果没有后续的处理，就

变成了死信，有死信自然就有了死信队列。

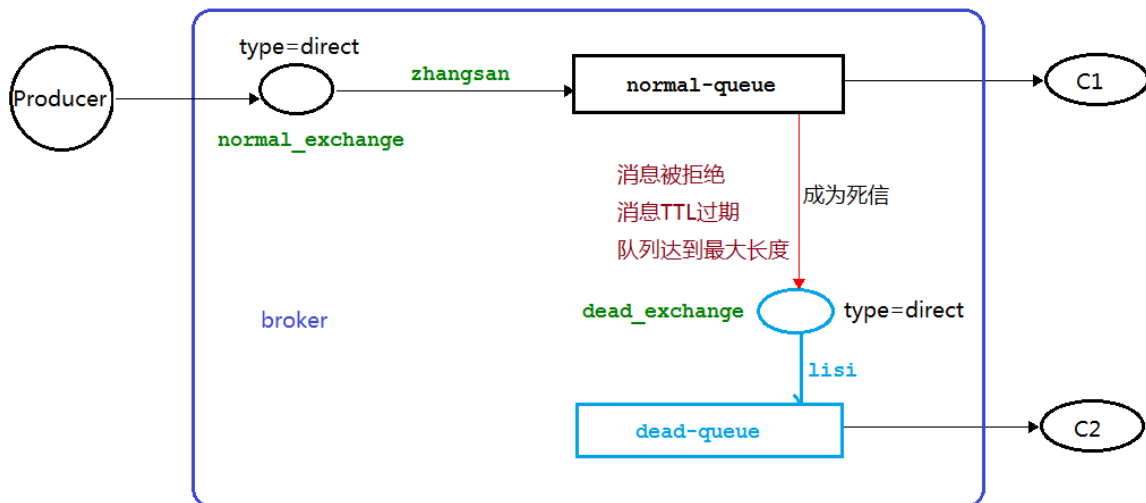
应用场景：为了保证订单业务的消息数据不丢失，需要使用到 RabbitMQ 的死信队列机制，当消息消费发生异常时，将消息投入死信队列中。还有比如说：用户在

商城下单成功并点击去支付后在指定时间未支付时自动失效

死信的来源

- 消息 TTL 过期
TTL是Time To Live的缩写, 也就是生存时间
- 队列达到最大长度
队列满了, 无法再添加数据到 mq 中
- 消息被拒绝
(basic.reject 或 basic.nack) 并且 requeue=false.

死信的实现



TTL

消费者1

```
1  /**
2   * @Author: ww
3   * @DateTime: 2022/4/30 20:15
4   * @Description: 死信队列 消费者1
5   */
6  public class Consumer01 {
7      // 普通交换机的名称
8      public static final String NORMAL_EXCHANGE = "normal_exchange";
9      // 死信交换机的名称
10     public static final String DEAD_EXCHANGE = "dead_exchange";
11     // 普通队列的名称
12     public static final String NORMAL_QUEUE = "normal_queue";
13     // 死信队列的名称
14     public static final String DEAD_QUEUE = "dead_queue";
15
16     public static void main(String[] args) throws Exception {
17         Channel channel = RabbitMQutils.getChannel();
18         // 声明死信交换机和死信交换机 类型为direct
```

```

19     channel.exchangeDeclare(NORMAL_EXCHANGE,
    BuiltInExchangeType.DIRECT);
20     channel.exchangeDeclare(DEAD_EXCHANGE, BuiltInExchangeType.DIRECT);
21
22     // 声明普通队列
23     Map<String, Object> arguments = new HashMap<>();
24     // 设置过期时间 10s 在生产者那边设置比较好
25     // arguments.put("x-message-ttl", 10000);
26     // 正常队列设置死信交换机
27     arguments.put("x-dead-letter-exchange", DEAD_EXCHANGE);
28     // 设置死信RoutingKey
29     arguments.put("x-dead-letter-routing-key", "lisi");
30     channel.queueDeclare(NORMAL_QUEUE, false, false, false, arguments);
31
32     // 声明死信队列
33     channel.queueDeclare(DEAD_QUEUE, false, false, false, null);
34
35     // 绑定普通的交换机与普通队列
36     channel.queueBind(NORMAL_QUEUE, NORMAL_EXCHANGE, "zhangsan");
37     // 绑定死信的交换机与死信队列
38     channel.queueBind(DEAD_QUEUE, DEAD_EXCHANGE, "lisi");
39     System.out.println("等待接收消息.....");
40
41     DeliverCallback deliverCallback = (consumerTag, message) -> {
42         System.out.println("Consumer01接收的消息: " + new
    String(message.getBody(), "utf-8"));
43     };
44
45
46     channel.basicConsume(NORMAL_QUEUE, true, deliverCallback, consumerTag -
    > {});
47     }
48 }

```

生产者

```

1  /**
2   * @Author: ww
3   * @DateTime: 2022/4/30 20:36
4   * @Description: 死信队列 生产者
5   */
6  public class Producer {
7      // 普通交换机的名称
8      public static final String NORMAL_EXCHANGE = "normal_exchange";
9      public static void main(String[] args) throws Exception {
10         Channel channel = RabbitMQUtils.getChannel();
11         channel.exchangeDeclare(NORMAL_EXCHANGE,
    BuiltInExchangeType.DIRECT);
12         // 发送死信消息 设置TTL时间(time to live)
13         AMQP.BasicProperties properties = new
    AMQP.BasicProperties().builder().expiration("10000").build();
14         for (int i = 1; i < 11; i++) {
15             String message = "info" + i;
16             channel.basicPublish(NORMAL_EXCHANGE, "zhangsan", properties ,
    message.getBytes());
17             System.out.println("生产者发送消息: " + message);
18         }

```

```
19     }
20 }
```

启动 C1，之后关闭消费者，模拟其接收不到消息。再启动 Producer

生产者未发送消息

Overview				Messages			Message rates		
Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
dead-queue	classic		idle	0	0	0			
normal-queue	classic	TTL DLX DLK	idle	0	0	0			

生产者发送了10条消息 此时正常消息队列有10条未消费信息

Overview				Messages			Message rates		
Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
dead-queue	classic		idle	0	0	0			
normal-queue	classic	TTL DLX DLK	idle	10	0	10	0.00/s		

时间过去10秒 正常队列里面的消息由于没有被消费 消息进入死信队列

Overview				Messages			Message rates		
Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
dead-queue	classic		idle	10	0	10			
normal-queue	classic	TTL DLX DLK	idle	0	0	0	0.00/s		

消费者2

```
1 public class Consumer02 {
2     public static final String DEAD_QUEUE = "dead_queue";
3     public static void main(String[] args) throws Exception {
4         Channel channel = RabbitMQUtils.getChannel();
5         System.out.println("等待接收消息.....");
6         DeliverCallback deliverCallback = (consumerTag, message) -> {
7             System.out.println("Consumer02接收的消息: " + new
String(message.getBody(), "utf-8"));
8         };
9         channel.basicConsume(DEAD_QUEUE, true, deliverCallback, consumerTag ->
{});
10    }
11 }
```

Overview				Messages			Message rates		
Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
dead-queue	classic		idle	0	0	0		0.00/s	0.00/s
normal-queue	classic	TTL DLX DLK	idle	0	0	0	0.00/s		

等待接收死信队列消息.....

Consumer02接收死信队列的消息info1
Consumer02接收死信队列的消息info2
Consumer02接收死信队列的消息info3
Consumer02接收死信队列的消息info4
Consumer02接收死信队列的消息info5
Consumer02接收死信队列的消息info6
Consumer02接收死信队列的消息info7
Consumer02接收死信队列的消息info8
Consumer02接收死信队列的消息info9
Consumer02接收死信队列的消息info10

死信队列里面的消息被C2消费

队列达到最大长度

- 1、去掉生产者代码中的TTL属性
- 2、在消费者1中添加如下代码

```
1 // 设置正常队列的长度限制
2 arguments.put("x-max-length", 5);
```

注：启动消费者1前先把原来的正常队列删除，否则会报错

- 3、消费者2代码不变

启动消费者1，再关掉使其假死，再启动生产者，会发现：

Queues

▼ All queues (8)

Pagination

Page 1 of 1 - Filter: ☐ Regex ?

Overview				Messages			Message rates			+/-
Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack	
Q1	classic		idle	0	0	0	0.00/s	0.00/s	0.00/s	
Q2	classic		idle	0	0	0	0.00/s	0.00/s	0.00/s	
ack_queue	classic	D	idle	0	0	0	0.00/s	0.00/s	0.00/s	
console	classic		idle	0	0	0	0.00/s	0.00/s	0.00/s	
dead_queue	classic		idle	5	0	5		0.00/s	0.00/s	
disk	classic		idle	0	0	0	0.00/s	0.00/s	0.00/s	
hello	classic		idle	0	0	0	0.00/s	0.00/s	0.00/s	
normal_queue	classic	Lim DLX DLK	idle	5	0	5	0.00/s	0.00/s	0.00/s	

► Add a new queue

然后启动消费者2，会发现消费者2从死信队列中消费了5个消息

```
Producer x Consumer02 x Consum
D:\JDK\bin\java.exe "-javaagent:
SLF4J: Failed to load class "org
SLF4J: Defaulting to no-operatio
SLF4J: See http://www.slf4j.org/
等待接收消息.....
Consumer02接收的消息: info1
Consumer02接收的消息: info2
Consumer02接收的消息: info3
Consumer02接收的消息: info4
Consumer02接收的消息: info5
```

Queues

▼ All queues (8)

Pagination

Page 1 of 1 - Filter: ☐ Regex ?

Overview				Messages			Message rates			+/-
Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack	
Q1	classic		idle	0	0	0	0.00/s	0.00/s	0.00/s	
Q2	classic		idle	0	0	0	0.00/s	0.00/s	0.00/s	
ack_queue	classic	D	idle	0	0	0	0.00/s	0.00/s	0.00/s	
console	classic		idle	0	0	0	0.00/s	0.00/s	0.00/s	
dead_queue	classic		idle	0	0	0	0.00/s	0.00/s	0.00/s	
disk	classic		idle	0	0	0	0.00/s	0.00/s	0.00/s	
hello	classic		idle	0	0	0	0.00/s	0.00/s	0.00/s	
normal_queue	classic	Lim DLX DLK	idle	5	0	5	0.00/s	0.00/s	0.00/s	

消息被拒

- 1、消息生产者代码同上生产者一致
- 2、C1 消费者代码(启动之后关闭该消费者 模拟其接收不到消息)

拒收消息 "info5"

```
1 public class Consumer01 {
2     // 普通交换机的名称
3     public static final String NORMAL_EXCHANGE = "normal_exchange";
4     // 死信交换机的名称
5     public static final String DEAD_EXCHANGE = "dead_exchange";
6     // 普通队列的名称
7     public static final String NORMAL_QUEUE = "normal_queue";
8     // 死信队列的名称
9     public static final String DEAD_QUEUE = "dead_queue";
10
11     public static void main(String[] args) throws Exception {
12         Channel channel = RabbitMqUtils.getChannel();
13         // 声明死信交换机和死信交换机 类型为direct
14         channel.exchangeDeclare(NORMAL_EXCHANGE,
15             BuiltinExchangeType.DIRECT);
16         channel.exchangeDeclare(DEAD_EXCHANGE, BuiltinExchangeType.DIRECT);
17
18         // 声明普通队列
19         Map<String, Object> arguments = new HashMap<>();
20         // 设置过期时间 10s 在生产者那边设置比较好
21         // arguments.put("x-message-ttl", 10000);
22         // 正常队列设置死信交换机
23         arguments.put("x-dead-letter-exchange", DEAD_EXCHANGE);
24         // 设置死信RoutingKey
25         arguments.put("x-dead-letter-routing-key", "lisi");
26         // 设置正常队列的长度限制
27         // arguments.put("x-max-length", 5);
28         channel.queueDeclare(NORMAL_QUEUE, false, false, false, arguments);
29
30         // 声明死信队列
31         channel.queueDeclare(DEAD_QUEUE, false, false, false, null);
32
33         // 绑定普通的交换机与普通队列
```

```

33     channel.queueBind(NORMAL_QUEUE, NORMAL_EXCHANGE, "zhangsang");
34     // 绑定死信的交换机与死信队列
35     channel.queueBind(DEAD_QUEUE, DEAD_EXCHANGE, "lisi");
36     System.out.println("等待接收消息.....");
37
38     DeliverCallback deliverCallback = (consumerTag, message) -> {
39         String msg = new String(message.getBody(), "utf-8");
40         if (msg.equals("info5")) {
41             System.out.println("Consumer01接收的消息是" + msg + ": 此消息是
被拒绝的");
42
43             channel.basicReject(message.getEnvelope().getDeliveryTag(), false);
44             } else {
45                 System.out.println("Consumer01接收的消息: " + msg);
46                 channel.basicAck(message.getEnvelope().getDeliveryTag(),
false);
47             }
48         };
49         // 开启手动应答
50         channel.basicConsume(NORMAL_QUEUE, false, deliverCallback, consumerTag
-> {});
51     }

```

dead_queue	classic		idle	1	0	1		0.00/s	0.00/s
disk	classic		idle	0	0	0	0.00/s	0.00/s	0.00/s
hello	classic		idle	0	0	0	0.00/s	0.00/s	0.00/s
normal_queue	classic	DLX DLK	idle	0	0	0	0.00/s	0.00/s	0.00/s

```

等待接收消息.....
Consumer01接收的消息: info1
Consumer01接收的消息: info2
Consumer01接收的消息: info3
Consumer01接收的消息: info4
Consumer01接收的消息是info5: 此消息是被拒绝的
Consumer01接收的消息: info6
Consumer01接收的消息: info7
Consumer01接收的消息: info8
Consumer01接收的消息: info9
Consumer01接收的消息: info10

```

RabbitMQ-延迟队列

延迟队列的概念

延时队列,队列内部是有序的, 最重要的特性就体现在它的延时属性上, 延时队列中的元素是希望 在指定时间到了以后或之前取出和处理, 简单来说, 延时队列就

是用来存放需要在指定时间被处理的 元素的队列。

延迟队列的使用场景

1. 订单在十分钟之内未支付则自动取消
2. 新创建的店铺，如果在十天内都没有上传过商品，则自动发送消息提醒
3. 用户注册成功后，如果三天内没有登陆则进行短信提醒
4. 用户发起退款，如果三天内没有得到处理则通知相关运营人员
5. 预定会议后，需要在预定的时间点前十分钟通知各个与会人员参加会议

这些场景都有一个特点，需要在某个事件发生之后或者之前的指定时间点完成某一项任务，如：发生订单生成事件，在十分钟之后检查该订单支付状态，然后将

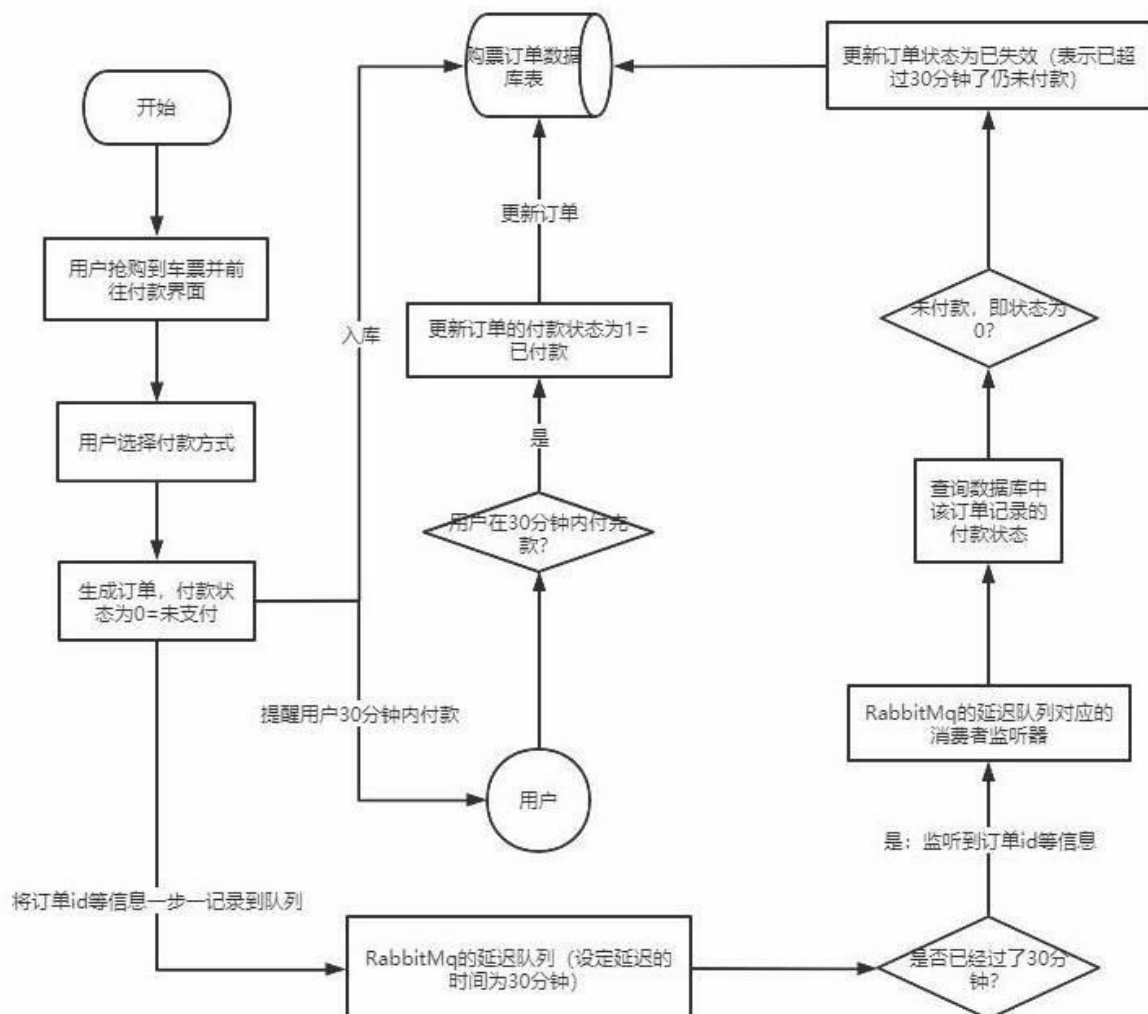
未支付的订单进行关闭；看起来似乎使用定时任务，一直轮询数据，每秒查一次，取出需要被处理的数据，然后处理不就完事了吗？如果数据量比较少，确实可

以这样做，比如：对于“如果账单一周内未支付则进行自动结算”这样的需求，如果对于时间不是严格限制，而是宽松意义上的一周，那么每天晚上跑个定时任务检

查一下所有未支付的账单，确实也是一个可行的方案。但对于数据量比较大，并且时效性较强的场景，如：“订单十分钟内未支付则关闭”，短期内未支付的订单数

据可能会有很多，活动期间甚至会达到百万甚至千万级别，对这么庞大的数据量仍旧使用轮询的方式显然是不可取的，很可能在一秒内无法完成所有订单的检

查，同时会给数据库带来很大压力，无法满足业务要求而且性能低下。



RabbitMQ中的TTL

TTL 是什么呢？TTL 是 RabbitMQ 中一个消息或者队列的属性，表明一条消息或者该队列中的所有消息的最大存活时间，单位是毫秒。

换句话说，如果一条消息设置了 TTL 属性或者进入了设置TTL 属性的队列，那么这条消息如果在 TTL 设置的时间内没有被消费，则会成为"死信"。如果同时配置了

队列的TTL 和消息的 TTL，那么较小的那个值将会被使用，有两种方式设置 TTL。

队列设置TTL

在创建队列的时候设置队列的"x-message-ttl"属性

```
1 //声明队列的 TTL
2 args.put("x-message-ttl", 10000);
3 return QueueBuilder.durable(QUEUE_A).withArguments(args).build();
```

消息设置TTL

```
1 rabbitTemplate.convertAndSend("x", "xc", message, correlationData ->{
2     // 延时时长
3     correlationData.getMessageProperties().setExpiration(ttlTime);
4     return correlationData;
5 });
```

两者的区别

如果设置了队列的 TTL 属性，那么一旦消息过期，就会被队列丢弃(如果配置了死信队列被丢到死信队列中)，而第二种方式，消息即使过期，也不一定会被马上丢

弃，因为消息是否过期是在即将投递到消费者之前判定的，如果当前队列有严重的消息积压情况，则已过期的消息也许还能存活较长时间；

另外，还需要注意的一点是，如果不设置 TTL，表示消息永远不会过期，如果将 TTL 设置为 0，则表示除非此时可以直接投递该消息到消费者，否则该消息将会被

丢弃。

整合springboot

1. 创建一个springboot模块
2. 添加依赖

```
1 <dependencies>
2     <!--RabbitMQ 依赖-->
3     <dependency>
4         <groupId>org.springframework.boot</groupId>
5         <artifactId>spring-boot-starter-amqp</artifactId>
6     </dependency>
7     <dependency>
8         <groupId>org.springframework.boot</groupId>
9         <artifactId>spring-boot-starter-web</artifactId>
10    </dependency>
11 </dependencies>
```

```

12         <groupId>org.springframework.boot</groupId>
13         <artifactId>spring-boot-starter-test</artifactId>
14         <scope>test</scope>
15     </dependency>
16     <dependency>
17         <groupId>com.alibaba</groupId>
18         <artifactId>fastjson</artifactId>
19         <version>1.2.47</version>
20     </dependency>
21     <dependency>
22         <groupId>org.projectlombok</groupId>
23         <artifactId>lombok</artifactId>
24     </dependency>
25     <!--swagger-->
26     <dependency>
27         <groupId>io.springfox</groupId>
28         <artifactId>springfox-swagger2</artifactId>
29         <version>2.9.2</version>
30     </dependency>
31     <dependency>
32         <groupId>io.springfox</groupId>
33         <artifactId>springfox-swagger-ui</artifactId>
34         <version>2.9.2</version>
35     </dependency>
36     <!--RabbitMQ 测试依赖-->
37     <dependency>
38         <groupId>org.springframework.amqp</groupId>
39         <artifactId>spring-rabbit-test</artifactId>
40         <scope>test</scope>
41     </dependency>
42 </dependencies>

```

3.修改配置文件

```

1 spring.rabbitmq.host=ipaddress
2 spring.rabbitmq.port=5672
3 spring.rabbitmq.username=admin
4 spring.rabbitmq.password=123

```

4.添加Swagger配置类

```

1 import org.springframework.context.annotation.Bean;
2 import org.springframework.context.annotation.Configuration;
3 import springfox.documentation.builders.ApiInfoBuilder;
4 import springfox.documentation.service.ApiInfo;
5 import springfox.documentation.service.Contact;
6 import springfox.documentation.spi.DocumentationType;
7 import springfox.documentation.spring.web.plugins.Docket;
8 import springfox.documentation.swagger2.annotations.EnableSwagger2;
9
10 /**
11  * @Author: ww
12  * @DateTime: 2022/5/2 20:30
13  * @Description: Swagger配置类
14  */
15

```

```

16 @Configuration
17 @EnableSwagger2
18 public class SwaggerConfig {
19     @Bean
20     public Docket webApiConfig(){
21         return new Docket(DocumentationType.SWAGGER_2)
22             .groupName("webApi")
23             .apiInfo(webApiInfo())
24             .select()
25             .build();
26     }
27     private ApiInfo webApiInfo(){
28         return new ApiInfoBuilder()
29             .title("rabbitmq 接口文档")
30             .description("本文档描述了 rabbitmq 微服务接口定义")
31             .version("1.0")
32             .contact(new Contact("itww", "https://bbbb.com",
33                 "123456@qq.com"))
34             .build();
35     }
36 }

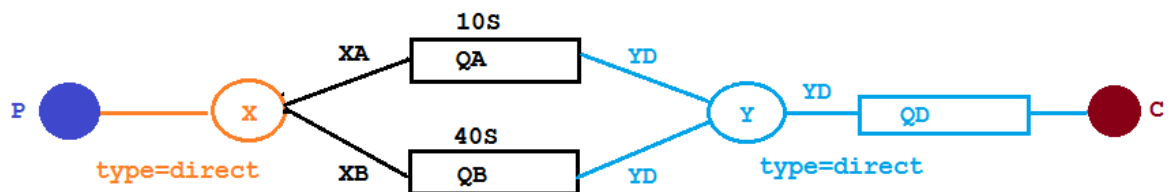
```

队列TTL

代码架构图

创建两个队列 QA 和 QB，两者队列 TTL 分别设置为 10S 和 40S，然后在创建一个交换机 X 和死信交换机 Y，它们的类型都是direct，创建一个死信队列 QD，它

们的绑定关系如下：



原先配置队列信息，写在了生产者和消费者代码中，现在可写在配置类中，生产者只发消息，消费者只接受消息

配置类

```

1  import org.springframework.amqp.core.*;
2  import org.springframework.beans.factory.annotation.Qualifier;
3  import org.springframework.context.annotation.Bean;
4  import org.springframework.context.annotation.Configuration;
5
6  import java.util.HashMap;
7  import java.util.Map;
8
9  /**
10   * @Author: ww
11   * @DateTime: 2022/5/2 20:36

```

```
12  * @Description: TTL队列 配置类
13  */
14  @Configuration
15  public class TtlQueueConfig {
16      // 普通交换机名称
17      public static final String X_EXCHANGE = "X";
18      // 死信交换机名称
19      public static final String Y_DEAD_LETTER_EXCHANGE = "Y";
20      // 普通队列名称
21      public static final String QUEUE_A = "QA";
22      public static final String QUEUE_B = "QB";
23      // 死信队列名称
24      public static final String DEAD_LETTER_QUEUE = "QD";
25
26      // 声明x交换机
27      @Bean("xExchange")
28      public DirectExchange xExchange(){
29          return new DirectExchange(X_EXCHANGE);
30      }
31
32      // 声明y交换机
33      @Bean("yExchange")
34      public DirectExchange yExchange(){
35          return new DirectExchange(Y_DEAD_LETTER_EXCHANGE);
36      }
37
38      // 声明普通队列A 10s
39      @Bean("queueA")
40      public Queue queueA(){
41          Map<String, Object> args = new HashMap<>(3);
42          //声明当前队列绑定的死信交换机
43          args.put("x-dead-letter-exchange", Y_DEAD_LETTER_EXCHANGE);
44          //声明当前队列的死信路由 key
45          args.put("x-dead-letter-routing-key", "YD");
46          //声明队列的 TTL
47          args.put("x-message-ttl", 10000);
48          return QueueBuilder.durable(QUEUE_A).withArguments(args).build();
49      }
50
51      // 声明普通队列B 40s
52      @Bean("queueB")
53      public Queue queueB(){
54          Map<String, Object> args = new HashMap<>(3);
55          //声明当前队列绑定的死信交换机
56          args.put("x-dead-letter-exchange", Y_DEAD_LETTER_EXCHANGE);
57          //声明当前队列的死信路由 key
58          args.put("x-dead-letter-routing-key", "YD");
59          //声明队列的 TTL
60          args.put("x-message-ttl", 40000);
61          return QueueBuilder.durable(QUEUE_B).withArguments(args).build();
62      }
63
64      // 声明死信队列QD
65      @Bean("queueD")
66      public Queue queueD(){
67          return QueueBuilder.durable(DEAD_LETTER_QUEUE).build();
68      }
69  }
```

```

70 // 绑定
71 @Bean
72 public Binding queueABindingX(@Qualifier("queueA") Queue queueA,
    @Qualifier("xExchange") DirectExchange xExchange){
73     return BindingBuilder.bind(queueA).to(xExchange).with("XA");
74 }
75 @Bean
76 public Binding queueBBindingX(@Qualifier("queueB") Queue queueB,
    @Qualifier("xExchange") DirectExchange xExchange){
77     return BindingBuilder.bind(queueB).to(xExchange).with("XB");
78 }
79 @Bean
80 public Binding queueDBindingY(@Qualifier("queueD") Queue queueD,
    @Qualifier("yExchange") DirectExchange xExchange){
81     return BindingBuilder.bind(queueD).to(xExchange).with("YD");
82 }
83 }

```

生产者

```

1 @GetMapping("/sendMsg/{message}")
2 public void sendMsg(@PathVariable String message){
3     log.info("当前时间: {},发送一条信息给两个TTL队列: {}", new
    Date().toString(), message);
4     rabbitTemplate.convertAndSend("x", "XA", "消息来自TTL为10s的队列: " +
    message);
5     rabbitTemplate.convertAndSend("x", "XB", "消息来自TTL为40s的队列: " +
    message);
6 }

```

消费者

```

1 @RabbitListener(queues = "QD")
2 public void received(Message message, Channel channel) throws Exception {
3     String msg = new String(message.getBody());
4     log.info("当前时间: {},收到死信队列信息: {}", new Date().toString(), msg);
5 }

```

发送请求: <http://localhost:8080/ttl/sendMsg/msg>

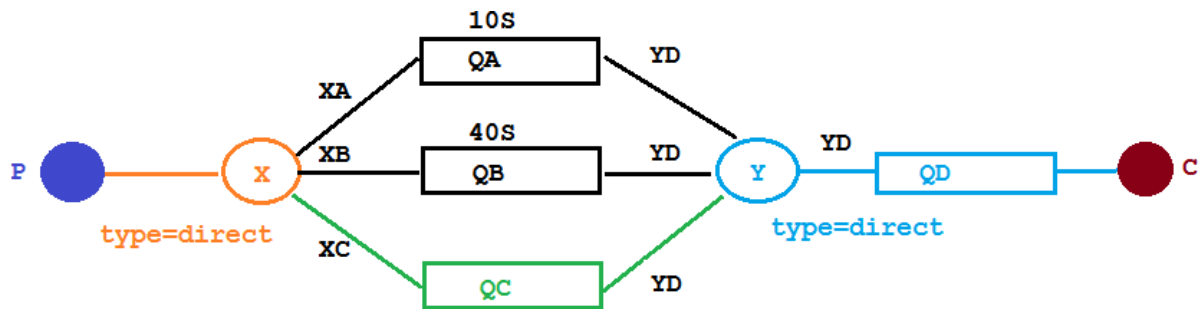
第一条消息在 10S 后变成了死信消息, 然后被消费者消费掉, 第二条消息在 40S 之后变成了死信消息, 然后被消费掉, 这样一个延时队列就打造完成了。

不过, 如果这样使用的话, 岂不是每增加一个新的时间需求, 就要新增一个队列, 这里只有 10S 和 40S 两个时间选项, 如果需要一个小时后处理, 那么就需要增

加TTL 为一个小时的队列, 如果是预定会议室然后提前通知这样的场景, 岂不是要增加无数个队列才能满足需求?

延迟队列TTL优化

在这里新增了一个队列 QC,绑定关系如下,该队列不设置TTL 时间



配置类添加

```
1 // 新的普通队列名称
2 public static final String QUEUE_C = "QC";
3 // 声明QC
4 @Bean
5 public Queue queueC(){
6     Map<String, Object> args = new HashMap<>(3);
7     //声明当前队列绑定的死信交换机
8     args.put("x-dead-letter-exchange", Y_DEAD_LETTER_EXCHANGE);
9     //声明当前队列的死信路由 key
10    args.put("x-dead-letter-routing-key", "YD");
11    //没有声明 TTL 属性
12    return QueueBuilder.durable(QUEUE_C).withArguments(args).build();
13 }
14 // 绑定
15 @Bean
16 public Binding queueCBindingX(@Qualifier("queueC") Queue queueC,
17 @Qualifier("xExchange") DirectExchange xExchange){
18     return BindingBuilder.bind(queueC).to(xExchange).with("XC");
19 }
```

生产者

```
1 @GetMapping("sendExpirationMsg/{message}/{ttlTime}")
2 public void sendMsg(@PathVariable String message,@PathVariable String
3 ttlTime) {
4     rabbitTemplate.convertAndSend("X", "XC", message, correlationData ->{
5         // 延时时长
6         correlationData.getMessageProperties().setExpiration(ttlTime);
7         return correlationData;
8     });
9     log.info("当前时间: {},发送一条时长{}毫秒 TTL 信息给队列 C:{}", new
10 Date(),ttlTime, message);
11 }
```

当前时间: Sun Sep 20 18:58:56 IRKT 2020,发送一条时长20000毫秒TTL信息给队列C:你好1
当前时间: Sun Sep 20 18:59:02 IRKT 2020,发送一条时长2000毫秒TTL信息给队列C:你好2
当前时间: Sun Sep 20 18:59:16 IRKT 2020,收到死信队列信息你好1
当前时间: Sun Sep 20 18:59:16 IRKT 2020,收到死信队列信息你好2

看起来似乎没什么问题,但是在最开始的时候,就介绍过如果使用在消息属性上设置 TTL 的方式,消息可能并不会按时“死亡”

因为 RabbitMQ 只会检查第一个消息是否过期，如果过期则丢到死信队列，如果第一个消息的延时时长很长，而第二个消息的延时时长很短，第二个消息并不会

优先得到执行。

这也就是为什么第二个延时2秒，却后执行。

RabbitMQ插件实现延迟队列

安装延时队列插件

1、在官网下载 <https://www.rabbitmq.com/community-plugins.html>，下载 rabbitmq_delayed_message_exchange 插件，然后解压放置到 RabbitMQ 的插件目录。

```
[root@ecs-288776 software]# ll
total 33616
-rw-r--r-- 1 root root 18850824 Apr 25 22:04 erlang-21.3-1.el7.x86_64.rpm
-rw-r--r-- 1 root root 43377 Apr 25 22:04 rabbitmq_delayed_message_exchange-3.8.0.ez
-rw-r--r-- 1 root root 15520399 Apr 25 22:04 rabbitmq-server-3.8.8-1.el7.noarch.rpm
[root@ecs-288776 software]# cp rabbitmq_delayed_message_exchange-3.8.0.ez /usr/lib/rabbitmq/lib/rabbitmq_server-3.8.8/plugins
[root@ecs-288776 software]# cd /usr/lib/rabbitmq/lib/rabbitmq_server-3.8.8/plugins
```

2、进入 RabbitMQ 的安装目录下的 plgins 目录，执行下面命令让该插件生效，然后重启 RabbitMQ

```
1 | rabbitmq-plugins enable rabbitmq_delayed_message_exchange
```

```
[root@ecs-288776 plugins]# rabbitmq-plugins enable rabbitmq_delayed_message_exchange

Enabling plugins on node rabbit@ecs-288776:
rabbitmq_delayed_message_exchange
The following plugins have been configured:
  rabbitmq_delayed_message_exchange
  rabbitmq_management
  rabbitmq_management_agent
  rabbitmq_web_dispatch
Applying plugin configuration to rabbit@ecs-288776...
The following plugins have been enabled:
  rabbitmq_delayed_message_exchange

started 1 plugins.
[root@ecs-288776 plugins]# systemctl restart rabbitmq-server
```

3、web界面查看效果

▼ Add a new exchange

Name: *

Type: ▼

Durability: ▼

Auto delete: ? ▼

Internal: ? ▼

Arguments: = String ▼

Add Alternate exchange ?

Add exchange

配置类

```
1  import org.springframework.amqp.core.Binding;
2  import org.springframework.amqp.core.BindingBuilder;
3  import org.springframework.amqp.core.CustomExchange;
4  import org.springframework.amqp.core.Queue;
5  import org.springframework.beans.factory.annotation.Qualifier;
6  import org.springframework.context.annotation.Bean;
7  import org.springframework.context.annotation.Configuration;
8
9  import java.util.HashMap;
10 import java.util.Map;
11
12 /**
13  * @Author: ww
14  * @DateTime: 2022/5/3 17:25
15  * @Description: TODO
16  */
17
18 @Configuration
19 public class DelayedQueueConfig {
20     // 队列
21     public static final String DELAYED_QUEUE_NAME = "delayed.queue";
22     // 交换机
23     public static final String DELAYED_EXCHANGE_NAME = "delayed.exchange";
24     // routing key
25     public static final String DELAYED_ROUTING_KEY = "delayed.routingkey";
26
27     // 声明队列
28     @Bean
29     public Queue delayedQueue() {
30         return new Queue(DELAYED_QUEUE_NAME);
31     }
32
33     // 声明交换机
34     @Bean
35     public CustomExchange delayedExchange(){
36         Map<String, Object> args = new HashMap<>();
37         args.put("x-delayed-type", "direct");
38         /**
39          * 参数1: 交换机名称
40          * 参数2: 交换机类型
41          * 参数3: 持久化
42          * 参数4: 自动删除
43          * 参数5: 其他参数
44          */
45         return new CustomExchange(DELAYED_EXCHANGE_NAME, "x-delayed-
message", true, false, args);
46     }
47
48     // 绑定
49     @Bean
50     public Binding bindingDelayedQueue(@Qualifier("delayedQueue") Queue
queue, @Qualifier("delayedExchange") CustomExchange delayedExchange) {
51         return
BindingBuilder.bind(queue).to(delayedExchange).with(DELAYED_ROUTING_KEY).noa
rgs();
}
```

```
52     }
53 }
```

生产者

```
1 @GetMapping("sendDelayMsg/{message}/{delayTime}")
2 public void sendMsg(@PathVariable String message,@PathVariable Integer
3 delayTime) {
4     rabbitTemplate.convertAndSend(DELAYED_EXCHANGE_NAME,
5     DELAYED_ROUTING_KEY, message, correlationData ->{
6
7     correlationData.getMessageProperties().setDelay(delayTime);
8     return correlationData;
9     });
10    log.info("当前时间: {}, 发送一条延迟 {} 毫秒的信息给队列 delayed.queue:{}",
11    new Date(),delayTime, message);
12 }
```

消费者

```
1 /**
2  * @Author: ww
3  * @DateTime: 2022/5/3 17:48
4  * @Description: 消费者 基于插件的延迟消息
5  */
6 @Slf4j
7 @Component
8 public class DelayQueueConsumer {
9     public static final String DELAYED_QUEUE_NAME = "delayed.queue";
10    @RabbitListener(queues = DELAYED_QUEUE_NAME)
11    public void receivedDelayedQueue(Message message){
12        String msg = new String(message.getBody());
13        log.info("当前时间: {},收到延时队列的消息: {}", new Date().toString(),
14        msg);
15    }
16 }
```

发送请求:

- <http://localhost:8080/ttl/sendDelayMsg/hello1/20000>(opens new window)
- <http://localhost:8080/ttl/sendDelayMsg/hello2/2000>

```
2022-05-05 18:57:02.743 INFO 67016 --- [nio-8080-exec-1] c.i.r.s.controller.SendMsgController : 当前时间: Thu May 05 18:57:02 CST 2022, 发送一条延迟 20000 毫秒的信息给队列 delayed.queue:hello1
2022-05-05 18:57:02.765 ERROR 67016 --- [nctionFactory1] c.i.r.s.config.MyCallBack : 消息(Body:'hello1' MessageProperties {headers={}, contentType=text/plain, contentEncoding=UTF-8, co
2022-05-05 18:57:02.765 INFO 67016 --- [nctionFactory1] c.i.r.s.config.MyCallBack : 交换机已经收到 id 为: 的消息
2022-05-05 18:57:04.483 INFO 67016 --- [nio-8080-exec-3] c.i.r.s.controller.SendMsgController : 当前时间: Thu May 05 18:57:04 CST 2022, 发送一条延迟 2000 毫秒的信息给队列 delayed.queue:hello2
2022-05-05 18:57:04.505 ERROR 67016 --- [nctionFactory2] c.i.r.s.config.MyCallBack : 消息(Body:'hello2' MessageProperties {headers={}, contentType=text/plain, contentEncoding=UTF-8, co
2022-05-05 18:57:04.505 INFO 67016 --- [nctionFactory2] c.i.r.s.config.MyCallBack : 交换机已经收到 id 为: 的消息
2022-05-05 18:57:06.512 INFO 67016 --- [ntContainer#2-1] c.i.r.s.consumer.DelayQueueConsumer : 当前时间: Thu May 05 18:57:06 CST 2022,收到延时队列的消息: hello2
2022-05-05 18:57:22.777 INFO 67016 --- [ntContainer#2-1] c.i.r.s.consumer.DelayQueueConsumer : 当前时间: Thu May 05 18:57:22 CST 2022,收到延时队列的消息: hello1
```

第二个消息被先消费掉了, 符合预期

总结

延时队列在需要延时处理的场景下非常有用，使用 RabbitMQ 来实现延时队列可以很好的利用 RabbitMQ 的特性，如：消息可靠发送、消息可靠投递、死信队列

来保障消息至少被消费一次以及未被正确处理的消息不会被丢弃。另外，通过 RabbitMQ 集群的特性，可以很好的解决单点故障问题，不会因为单个节点挂掉导致

致延时队列不可用或者消息丢失。

当然，延时队列还有很多其它选择，比如利用 Java 的 DelayQueue，利用 Redis 的 zset，利用 Quartz 或者利用 kafka 的时间轮，这些方式各有特点,看需要适用

的场景。

RabbitMQ-发布确认高级

在生产环境中由于一些不明原因，导致 rabbitmq 重启，在 RabbitMQ 重启期间生产者消息投递失败，导致消息丢失，需要手动处理和恢复。于是，我们开始思

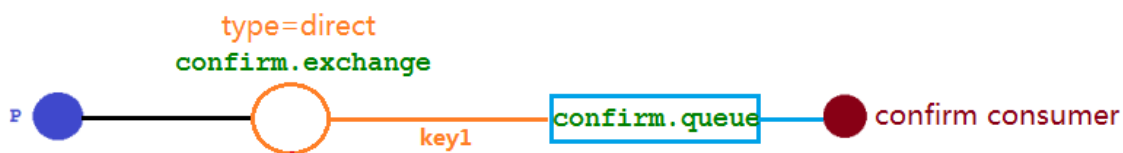
考，如何才能进行 RabbitMQ 的消息可靠投递呢？特别是在这样比较极端的情况，RabbitMQ 集群不可用的时候，无法投递的消息该如何处理呢？

发布确认springboot版本

确认机制方案



代码架构图：



在配置文件当中需要添加

```
1 | spring.rabbitmq.publisher-confirm-type=correlated
```

- `NONE` 值是禁用发布确认模式，是默认值
- `CORRELATED` 值是发布消息成功到交换器后会触发回调方法
- `SIMPLE` 值经测试有两种效果，其一效果和 `CORRELATED` 值一样会触发回调方法，其二在发布消息成功后使用 `rabbitTemplate` 调用 `waitForConfirms` 或

waitForConfirmsOrDie 方法等待 broker 节点返回发送结果，根据返回结果来判定下一步的逻辑，要注意的点是 waitForConfirmsOrDie 方法如果返回 false

则会关闭 channel，则接下来无法发送消息到 broker;

实现

配置类

```
1 import org.springframework.amqp.core.*;
2 import org.springframework.beans.factory.annotation.Qualifier;
3 import org.springframework.context.annotation.Bean;
4 import org.springframework.context.annotation.Configuration;
5
6 /**
7  * @Author: ww
8  * @DateTime: 2022/5/3 20:02
9  * @Description: 发布确认(高级)
10 */
11 @Configuration
12 public class ConfirmConfig {
13     // 交换机
14     public static final String CONFIRM_EXCHANGE_NAME = "confirm_exchange";
15     // 队列
16     public static final String CONFIRM_QUEUE_NAME = "confirm_queue";
17     // routing key
18     public static final String CONFIRM_ROUTING_KEY = "key1";
19
20     @Bean("confirmExchange")
21     public DirectExchange confirmExchange(){
22         return
23         ExchangeBuilder.directExchange(CONFIRM_EXCHANGE_NAME).durable(true).withArgument("alternate-exchange", BACKUP_EXCHANGE_NAME).build();
24     }
25
26     @Bean("confirmQueue")
27     public Queue confirmQueue(){
28         return QueueBuilder.durable(CONFIRM_QUEUE_NAME).build();
29     }
30
31     @Bean
32     public Binding queueBindingExchange(@Qualifier("confirmQueue") Queue queue, @Qualifier("confirmExchange") DirectExchange exchange) {
33         return BindingBuilder.bind(queue).to(exchange).with("key1");
34     }
35 }
```

消息生产者的回调接口

```
1 import lombok.extern.slf4j.Slf4j;
2 import org.springframework.amqp.core.Message;
3 import org.springframework.amqp.core.ReturnedMessage;
4 import org.springframework.amqp.rabbit.connection.CorrelationData;
5 import org.springframework.amqp.rabbit.core.RabbitTemplate;
6 import org.springframework.beans.factory.annotation.Autowired;
7 import org.springframework.stereotype.Component;
8
9 import javax.annotation.PostConstruct;
```

```

10
11 /**
12  * @Author: ww
13  * @DateTime: 2022/5/3 20:36
14  * @Description: TODO
15  */
16 @Slf4j
17 @Component //1
18 public class MyCallback implements RabbitTemplate.ConfirmCallback {
19
20     @Autowired //2
21     private RabbitTemplate rabbitTemplate;
22
23     @PostConstruct //3
24     public void init(){
25         // 注入
26         rabbitTemplate.setConfirmCallback(this);
27     }
28
29     /**
30      * 交换机确认回调方法
31      * 1.发消息 交换机接收到了 回调
32      *   1.1 correlationData 保存回调消息的ID及相关信息
33      *   1.2 交换机接收到消息 ack = true
34      *   1.3 cause: null
35      * 2.发消息 交换机接收失败了 回调
36      *   2.1 correlationData 保存回调消息的ID及相关信息
37      *   2.2 交换机接收到消息 ack = false
38      *   2.3 cause: 失败的原因
39      */
40     @Override
41     public void confirm(CorrelationData correlationData, boolean ack, String
cause) {
42         String id = correlationData != null ? correlationData.getId() : "";
43         if(ack){
44             log.info("交换机已经收到 id 为:{}]的消息",id);
45         }else{
46             log.info("交换机还未收到 id 为:{}]消息,由于原因:{}",id,cause);
47         }
48     }
49 }
50 }

```

生产者

```

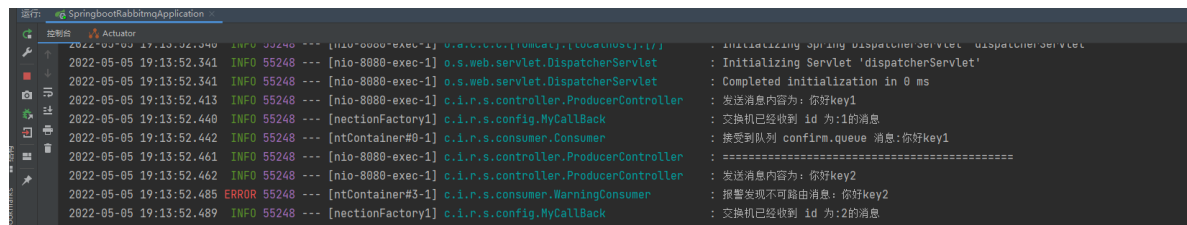
1 @GetMapping("/sendMessage/{message}")
2 public void sendMessage(@PathVariable String message){
3     CorrelationData correlationData = new CorrelationData("1");
4     rabbitTemplate.convertAndSend(ConfirmConfig.CONFIRM_EXCHANGE_NAME,
ConfirmConfig.CONFIRM_ROUTING_KEY, message+"key1", correlationData);
5     log.info("发送消息内容为: {}", message+"key1");
6 }

```

消费者

```
1 public static final String CONFIRM_QUEUE_NAME = "confirm_queue";
2 @RabbitListener(queues = ConfirmConfig.CONFIRM_QUEUE_NAME)
3 public void receiveConfirmMessage(Message message) {
4     String msg = new String(message.getBody());
5     log.info("接受到队列 confirm.queue 消息:{}", msg);
6 }
```

访问访问: <http://localhost:8080/confirm/sendMessage/你好>



可以看到, 发送了两条消息, 第一条消息的 RoutingKey 为 "key1", 第二条消息的 RoutingKey 为 "key2", 两条消息都成功被交换机接收, 也收到了交换机的确认回调, 但消费者只收到了一条消息, 因为 第二条消息的 RoutingKey 与队列的 BindingKey 不一致, 也没有其它队列能接收这个消息, 所有第二条消息被直接丢弃了。

回退消息

Mandatory 参数

在仅开启了生产者确认机制的情况下, 交换机接收到消息后, 会直接给消息生产者发送确认消息, 如果发现该消息不可路由, 那么消息会被直接丢弃, 此时生产者

是不知道消息被丢弃这个事件的。

那么如何让无法被路由的消息帮我想办法处理一下? 最起码通知我一声, 我好自己处理啊。通过设置 mandatory 参数可以在当消息传递过程中不可达目的地时将

消息返回给生产者。

添加配置文件

```
1 #消息退回
2 spring.rabbitmq.publisher-returns=true
```

修改回调接口

```
1 /**
2  * @Author: ww
3  * @DateTime: 2022/5/3 20:36
4  * @Description: TODO
5  */
6 @Slf4j
7 @Component //1
8 public class MyCallBack implements RabbitTemplate.ConfirmCallback,
9     RabbitTemplate.ReturnsCallback {
10
11     @Autowired //2
12     private RabbitTemplate rabbitTemplate;
```

```

12
13     @PostConstruct //3
14     public void init(){
15         // 注入
16         rabbitTemplate.setConfirmCallback(this);
17         rabbitTemplate.setReturnsCallback(this );
18     }
19
20     /**
21     * 交换机确认回调方法
22     * 1.发消息 交换机接收到了 回调
23     *   1.1 correlationData 保存回调消息的ID及相关信息
24     *   1.2 交换机接收到消息 ack = true
25     *   1.3 cause: null
26     * 2.发消息 交换机接收失败了 回调
27     *   2.1 correlationData 保存回调消息的ID及相关信息
28     *   2.2 交换机接收到消息 ack = false
29     *   2.3 cause: 失败的原因
30     */
31     @Override
32     public void confirm(CorrelationData correlationData, boolean ack, String
cause) {
33         String id = correlationData != null ? correlationData.getId() : "";
34         if(ack){
35             log.info("交换机已经收到 id 为: {}的消息",id);
36         }else{
37             log.info("交换机还未收到 id 为: {}消息,由于原因: {}",id,cause);
38         }
39     }
40 }
41
42 // 可以在消息传递过程中不可达目的地时将消息返回给生产者 只有不可达目的地时才进行回退
43 @Override
44 public void returnedMessage(ReturnedMessage returned) {
45     log.error("消息 {}, 被交换机 {} 退回, 原因: {}, 路由key:
{}",returned.getMessage().toString(), returned.getExchange(),
returned.getReplyText(), returned.getRoutingKey());
46 }
47 }

```

备份交换机

有了 mandatory 参数和回退消息，我们获得了对无法投递消息的感知能力，有机会在生产者的消息无法被投递时发现并处理。但有时候，我们并不知道该如何处

理这些无法路由的消息，最多打个日志，然后触发报警，再来手动处理。而通过日志来处理这些无法路由的消息是很不优雅的做法，特别是当生产者所在的服务

有多台机器的时候，手动复制日志会更加麻烦而且容易出错。而且设置 mandatory 参数会增加生产者的复杂性，需要添加处理这些被退回的消息的逻辑。如果既

不想丢失消息，又不想增加生产者的复杂性，该怎么做呢？前面在设置死信队列的文章中，我们提到，可以为队列设置死信交换机来存储那些处理失败的消息，

可是这些不可路由消息根本没有机会进入到队列，因此无法使用死信队列来保存消息。在 RabbitMQ 中，有一种备份交换机的机制存在，可以很好的应对这个问

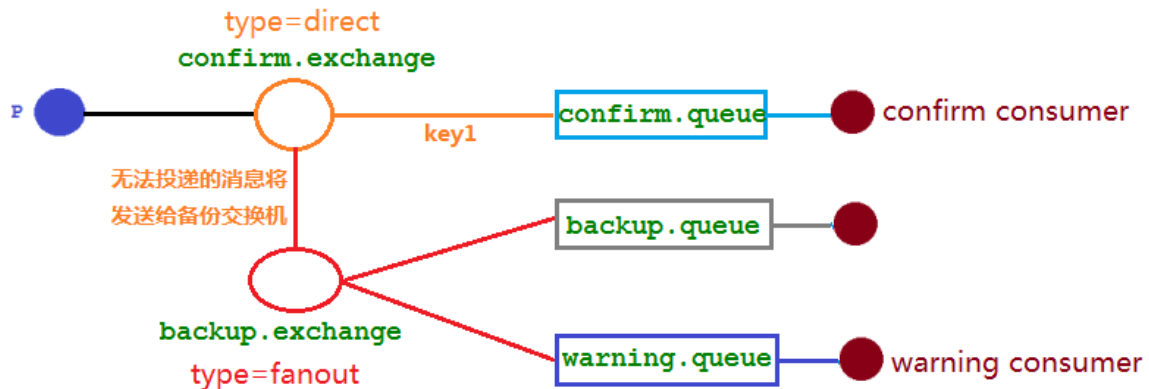
题。什么是备份交换机呢？备份 交换机可以理解为 RabbitMQ 中交换机的“备胎”，当我们为某一个交换机声明一个对应的备份交换机时， 就是为它创建一个备

胎，当交换机接收到一条不可路由消息时，将会把这条消息转发到备份交换机中，由 备份交换机来进行转发和处理，通常备份交换机的类型为 Fanout ， 这样就能

把所有消息都投递到与其绑 定的队列中，然后我们在备份交换机下绑定一个队列，这样所有那些原交换机无法被路由的消息，就会都 进入这个队列了。当然，我

们还可以建立一个报警队列，用独立的消费者来进行监测和报警。

代码架构图



配置类

```
1 import org.springframework.amqp.core.*;
2 import org.springframework.beans.factory.annotation.Qualifier;
3 import org.springframework.context.annotation.Bean;
4 import org.springframework.context.annotation.Configuration;
5
6 /**
7  * @Author: ww
8  * @DateTime: 2022/5/3 20:02
9  * @Description: 发布确认(高级)
10  */
11 @Configuration
12 public class ConfirmConfig {
13     // 交换机
14     public static final String CONFIRM_EXCHANGE_NAME = "confirm_exchange";
15     // 队列
16     public static final String CONFIRM_QUEUE_NAME = "confirm_queue";
17     // routing key
18     public static final String CONFIRM_ROUTING_KEY = "key1";
19
20     // 备份交换机
21     public static final String BACKUP_EXCHANGE_NAME = "backup_exchange";
22     // 备份队列
23     public static final String BACKUP_QUEUE_NAME = "backup_queue";
24     // 报警队列
25     public static final String WARNING_QUEUE_NAME = "warning_queue";
26
27
28     @Bean("confirmExchange")
29     public DirectExchange confirmExchange(){
```

```

30         return
ExchangeBuilder.directExchange(CONFIRM_EXCHANGE_NAME).durable(true).withArgument("alternate-exchange", BACKUP_EXCHANGE_NAME).build();
31     }
32
33     @Bean("confirmQueue")
34     public Queue confirmQueue(){
35         return QueueBuilder.durable(CONFIRM_QUEUE_NAME).build();
36     }
37
38     @Bean
39     public Binding queueBindingExchange(@Qualifier("confirmQueue") Queue queue, @Qualifier("confirmExchange") DirectExchange exchange) {
40         return BindingBuilder.bind(queue).to(exchange).with("key1");
41     }
42
43
44     // 备份交换机
45     @Bean("backupExchange")
46     public FanoutExchange backupExchange(){
47         return new FanoutExchange(BACKUP_EXCHANGE_NAME);
48     }
49
50     @Bean("backupQueue")
51     public Queue backupQueue(){
52         return QueueBuilder.durable(BACKUP_QUEUE_NAME).build();
53     }
54
55     @Bean("warningQueue")
56     public Queue warningQueue(){
57         return QueueBuilder.durable(WARNING_QUEUE_NAME).build();
58     }
59
60     @Bean
61     public Binding
backupQueueBindingBackupExchange(@Qualifier("backupQueue") Queue queue,
@Qualifier("backupExchange") FanoutExchange exchange) {
62         return BindingBuilder.bind(queue).to(exchange);
63     }
64     @Bean
65     public Binding
warningQueueBindingBackupExchange(@Qualifier("warningQueue") Queue queue,
@Qualifier("backupExchange") FanoutExchange exchange) {
66         return BindingBuilder.bind(queue).to(exchange);
67     }
68 }

```

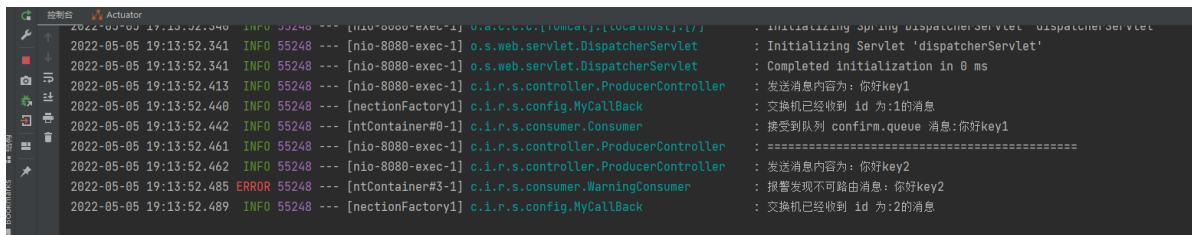
报警消费者

```

1  @Slf4j
2  @Component
3  public class WarningConsumer {
4      // 接收报警信息
5      @RabbitListener(queues = ConfirmConfig.WARNING_QUEUE_NAME)
6      public void receiveWarningMsg(Message message){
7          String msg = new String(message.getBody());
8          log.error("报警发现不可路由消息: {}", msg);
9      }
10 }

```

访问: <http://localhost:8080/confirm/sendMessage/你好>



```

2022-05-05 19:13:52.340 INFO 55248 --- [nio-8080-exec-1] o.s.b.c.c.tomcat.[tomcat].[localhost].[] : Initializing Spring DispatcherServlet 'dispatcherServlet'
2022-05-05 19:13:52.341 INFO 55248 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2022-05-05 19:13:52.413 INFO 55248 --- [nio-8080-exec-1] c.i.r.s.controller.ProducerController : Completed initialization in 0 ms
2022-05-05 19:13:52.440 INFO 55248 --- [nio-8080-exec-1] c.i.r.s.controller.ProducerController : 发送消息内容为: 你好key1
2022-05-05 19:13:52.442 INFO 55248 --- [ntContainer#0-1] c.i.r.s.consumer.Consumer : 交换机已经收到 id 为:1的消息
2022-05-05 19:13:52.461 INFO 55248 --- [nio-8080-exec-1] c.i.r.s.controller.ProducerController : 接收到队列 confirm.queue 消息:你好key1
2022-05-05 19:13:52.462 INFO 55248 --- [nio-8080-exec-1] c.i.r.s.controller.ProducerController : =====
2022-05-05 19:13:52.485 ERROR 55248 --- [ntContainer#3-1] c.i.r.s.consumer.WarningConsumer : 发送消息内容为: 你好key2
2022-05-05 19:13:52.489 INFO 55248 --- [ntContainer#3-1] c.i.r.s.consumer.WarningConsumer : 报警发现不可路由消息: 你好key2
2022-05-05 19:13:52.489 INFO 55248 --- [ntContainer#3-1] c.i.r.s.config.MyCallBack : 交换机已经收到 id 为:2的消息

```

mandatory 参数与备份交换机可以一起使用的时候, 如果两者同时开启, 消息究竟何去何从? 谁优先级高, 经过上面结果显示答案是**备份交换机优先级高**。

RabbitMQ-幂等性、优先级、惰性

幂等性

概念

用户对于同一操作发起的一次请求或者多次请求的结果是一致的, 不会因为多次点击而产生了副作用。举个最简单的例子, 那就是支付, 用户购买商品后支付,

支付扣款成功, 但是返回结果的时候网络异常, 此时钱已经扣了, 用户再次点击按钮, 此时会进行第二次扣款, 返回结果成功, 用户查询余额发现多扣钱了, 流

水记录也变成了两条。在以前的单应用系统中, 我们只需要把数据操作放入事务中即可, 发生错误立即回滚, 但是再响应客户端的时候也有可能出现网络中断或

者异常等等。

消息重复消费

消费者在消费 MQ 中的消息时, MQ 已把消息发送给消费者, 消费者在给 MQ 返回 ack 时网络中断, 故 MQ 未收到确认信息, 该条消息会重新发给其他的消费者, 或者在网络重连后再次发送给该消费者, 但实际上该消费者已成功消费了该条消息, 造成消费者消费了重复的消息。

解决思路

MQ 消费者的幂等性的解决一般使用全局 ID 或者写个唯一标识比如时间戳 或者 UUID 或者订单消费者消费 MQ 中的消息也可利用 MQ 的该 id 来判断, 或者可按

自己的规则生成一个全局唯一 id, 每次消费消息时用该 id 先判断该消息是否已消费过。

消费端的幂等性保障

在海量订单生成的业务高峰期，生产端有可能就会重复发生了消息，这时候消费端就要实现幂等性，这就意味着我们的消息永远不会被消费多次，即使我们收到

了一样的消息。

业界主流的幂等性有两种操作:a. 唯一 ID+指纹码机制,利用数据库主键去重, b.利用 redis 的原子性去实现

- 唯一ID+指纹码机制

指纹码：我们的一些规则或者时间戳加别的服务给到的唯一信息码,它并不一定是我们系统生成的，基本都是由我们的业务规则拼接而来，但是一定要保证唯一性，

然后就利用查询语句进行判断这个 id 是否存在数据库中，优势就是实现简单就一个拼接，然后查询判断是否重复；劣势就是在高并发时，如果是单个数据库就会

有写入性能瓶颈当然也可以采用分库分表提升性能，但也不是我们最推荐的方式。

- note Redis 原子性

利用 redis 执行 setnx 命令，天然具有幂等性。从而实现不重复消费

优先级队列

使用场景

在我们系统中有一个订单催付的场景，我们的客户在天猫下的订单,淘宝会及时将订单推送给我们，如果在用户设定的时间内未付款那么就会给用户推送一条短信

提醒，很简单的一个功能对吧，但是，tmall 商家对我们来说，肯定是要分大客户和小客户的对吧，比如像苹果，小米这样大商家一年起码能给我们创造很大的利

润，所以理应当然，他们的订单必须得到优先处理，而曾经我们的后端系统是使用 redis 来存放的定时轮询，大家都知道 redis 只能用 List 做一个简简单单的消息

队列，并不能实现一个优先级的场景，所以订单量大了后采用 RabbitMQ 进行改造和优化,如果发现是大客户的订单给一个相对比较高的优先级，否则就是默认优

先级。

如何添加

1.控制台添加

▼ Add a new queue

Type:

Name:

Durability:

Auto delete:

Arguments: =

=

Add

2.队列中代码添加优先级

```
1 // 设置参数
2 Map<String, Object> arguments = new HashMap<>();
3 arguments.put("x-max-priority", 10); // 官方允许0-255
4 channel.queueDeclare(QueueName, true, false, false, arguments);
```

3.消息中代码添加优先级

```
1 AMQP.BasicProperties properties = new
  AMQP.BasicProperties().builder().priority(5).build();
```

4.注意事项

要让队列实现优先级需要做的事情有如下事情: 队列需要设置为优先级队列, 消息需要设置消息的优先级, 消费者需要等待消息已经发送到队列中才去消费因

为, 这样才有机会对消息进行排序。

惰性队列

使用场景

RabbitMQ 从 3.6.0 版本开始引入了惰性队列的概念。惰性队列会尽可能的将消息存入磁盘中, 而在消费者消费到相应的消息时才会被加载到内存中, 它的一个重

要的设计目标是能够支持更长的队列, 即支持更多的消息存储。当消费者由于各种各样的原因(比如消费者下线、宕机亦或者是由于维护而关闭等)而致使长时间内

不能消费消息造成堆积时, 惰性队列就很有必要了。

默认情况下, 当生产者将消息发送到 RabbitMQ 的时候, 队列中的消息会尽可能的存储在内存之中, 这样可以更加快速的将消息发送给消费者。即使是持久化的

消息, 在被写入磁盘的同时也会在内存中驻留一份备份。当RabbitMQ 需要释放内存的时候, 会将内存中的消息换页至磁盘中, 这个操作会耗费较长的时间, 也会

阻塞队列的操作, 进而无法接收新的消息。虽然 RabbitMQ 的开发者们一直在升级相关的算法, 但是效果始终不太理想, 尤其是在消息量特别大的时候。

两种模式

队列具备两种模式: default 和 lazy。默认的为default 模式, 在3.6.0 之前的版本无需做任何变更。lazy 模式即为惰性队列的模式, 可以通过调用

channel.queueDeclare 方法的时候在参数中设置, 也可以通过 Policy 的方式设置, 如果一个队列同时使用这两种方式设置的话, 那么 Policy 的方式具备更高的优

先级。如果要通过声明的方式改变已有队列的模式的话, 那么只能先删除队列, 然后再重新声明一个新的。

在队列声明的时候可以通过"x-queue-mode"参数来设置队列的模式, 取值为"default"和"lazy"。下面示例中演示了一个惰性队列的声明细节:

```
1 Map<String, Object> args = new HashMap<String, Object>();
2 args.put("x-queue-mode", "lazy");
3 channel.queueDeclare("myqueue", false, false, false, args);
```

内存开销对比

Number of messages	Message body size	Message type	Producers	Consumers
1,000,000	1,000 bytes	persistent	1	0

The RAM utilization for default & lazy queues **after** ingesting the above messages:

Queue mode	Queue process memory	Messages in memory	Memory used by messages	Node memory
default	257 MB	386,307	368 MB	734 MB
lazy	159 KB	0	0	117 MB

Both queues persisted 1,000,000 messages and used 1.2 GB of disk space.

在发送 1 百万条消息，每条消息大概占 1KB 的情况下，普通队列占用内存是 1.2GB，而惰性队列仅仅占用 1.5MB