什么是 MQ?

mq 是一个消息队列,其主要目的是为了解决传统的消息传输上管理困难,效率不高的问题.

mq 有三大优点:解耦,异步,削峰.

- 解耦: 如果是传统的消息通讯方式,无论是哪一方都要去维护一份供外部通讯的这个一个接口,而且各方处理消息的能力有限,效率上明显跟不上,并且这样子二者之间的耦合度非常高,对于拓展管理方面极不友好,而是要了 mq 就不一样,发送方只需要将消息发送给 mq 就可以了,别的不用考虑,接口什么的由 mq 去维护,接收方也只需要去 mq 里消费消息就可以了,就需要其他开销,一切由 mq 中间件来做,达到了解耦操作.
- 异步: 使用 mq,还可以达到异步效果,极大地提升了消息传输的效率.发送方在发送消息后不需要关心消费方是否能消费完成,还可以继续发送其他消息.
- 削峰:如果是传统的消息通讯,一下子有大量的消息发送给接收方,这样对于接收方的处理压力是很大的,而我们刚好可以利用 mq 达到一个缓冲操作,一旦流量超出了接收方处理范围,不用担心,只需要慢慢消费即可,像经典的双十一,就很容易会使用到 mq 这么一个优点.

虽然 mq 有三大优点,但是我们还是得关心其一些缺点:

因为增加了中间件,系统复杂度肯定大大提高,增加了很多维护的成本,比如我们要保证消息不丢失(一致性)和消息幂等性问题,还要保证 mq 的高可用等.

MQ 有什么缺点?

系统可用性降低

系统引入的外部依赖越多，越容易挂掉，本来你就是 A 系统调用 BCD 三个系统的接口就好了，人 ABCD 四个系统好好的，没啥问题，你偏加个 MQ 进来，万一 MQ 挂了咋

整？MQ 挂了，整套系统崩溃了，你不就完了么。(可以利用集群解决)

系统复杂性提高

硬生生加个 MQ 进来，你怎么保证消息没有重复消费？怎么处理消息丢失的情况？怎么保证消息传递的顺序性？头大头大，问题一大堆，痛苦不已。

一致性问题(保证消息不丢失)

A 系统处理完了直接返回成功了，人都以为你这个请求就成功了；但是问题是，要是 BCD 三个系统那里，BD 两个系统写库成功了，结果 C 系统写库失败了，咋整？你这数据就不一致了。

所以消息队列实际是一种非常复杂的架构，你引入它有很多好处，但是也得针对它带来的坏处做各种额外的技术方案和架构来规避掉，最好之后，你会发现，妈呀，系统复杂度提升了一个数量级，也许是复杂了 10 倍。但是关键时刻，用，还是得用的。

谈谈你对 AMQP 的理解

AMQP 协议是什么?

AMQP 协议，所谓的高级消息队列协议，可以把它理解成一种公认的协议规范，就像 http 协议一样，只是这个 AMQP 协议针对的是消息队列。这个协议使得遵从了它的规范的客户端应用和消息中间件服务器的全功能互操作成为可能。

了解下 AMQP 协议的基本概念

- Broker：用于接受信息和分发信息的应用。
- Virtual hosts:在一个 Broker 上面划分出多个隔离的环境，这多个环境就可以理解成是 Virtual hosts，就像使用虚拟机一样，每个虚拟机之间都有完整的组件，各 Virtual hosts 下的用户、交换器以及队列等互不影响，这样方便不同的业务团队在使

用同一个 Rabbit server 提供的服务时，能够划清界限。

- Connection：消息生产者和消息消费者还有 Broker 之间的 TCP 连接，如果要断开连接，只会在客户端断开，而
- Broker 不会断开连接，除非网络出现了故障或者 Broker 服务出了问题。
- Channel：通道，如果每一次访问消息队列中间件都建立一个 TCP 连接的话，那么系统资源会被大量的占用，效率也会降低，所以 AMQP 提供了 Channel 机制，共享同一个 TCP 连接，而一个 TCP 连接里可以有大量的 Channel。假设如果有多个线程访问消息队列中间件服务，每个线程通常都会有自己单独的 Channel 来做通信，而每个 Channel 会有自己的 Channel id，这样客户端和 Broker 就能够互相识别 Channel，所以 Channel 之间是完全隔离的。
- Exchange：交换机，这是消息到达 Broker 的第一站，由于 Exchange 和 Queues 之间有绑定键来确定双方发送消息的匹配规则，所以这时 Exchange 会根据消息的路由键和自己的类型，来匹配绑定规则，将消息分发到对应的 Queues 上。
- Queue：队列，消息所到达的最终站，消费者从这里拿消息做消费。
- Binding：可以把它理解成一个虚拟的连接，定义了 Exchange 和 Queues 之间的匹配规则，只有匹配这个规则的交换机里的消息才会被发送到这个队列里，不过如果消息没法找到匹配的队列的话，那么，根据该条消息的属性，这个消息要么被丢弃，要么返回生产者那里。

Exchange 的类型

这里把 Exchange 和 Queues 之间的匹配规则称之为绑定键。

类型说明

- direct(直连) 消息的路由键要和绑定键一模一样才能分发到队列

- fanout(广播) 无需绑定键，只要有和这个交换机做绑定的队列，都会收到消息，有点类似发布-订阅
- topic(主题) 这个类型的交换机要求消息路由键和绑定键要模糊匹配才能分发，以.号来分割每个词， #号代表匹配多个词，*号代表匹配只一个词
- headers(header 属性) 这种类型的交换机不再是基于路由键了，而是基于消息中的 header 属性，只有消息中 header 属性的值与绑定键相同时，消息才会被分发到相应的队列中

默认交换机 如果不指定上述的交换机类型，就会使用默认的 direct 类型，同时绑定键默认是队列名，所以消息会分发到与路由键同名的队列里

vhost 是什么? 起什么作用?

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

特性：

- 每一个 vhost 本质上是一个小型的独立的 rabbitmq 服务器，拥有自己独立的完整的一套队列、绑定关系、交换器等。同一个服务器上的多个 vhost 是完全隔离的。队列及交换器等不互通。
- 所以一个 broker 可以开设多个 vhost，用于不同用户的权限分离

如何创建 vhost?

- 通过前台页面的 admin 中创建
- 使用 rabbitmqctl add_vhost vhost 名称 命令

如何删除 vhost?

- 前台删除
- rabbitmqctl delete_vhost vhost_name

消息是如何路由的?

消息提供方->路由->一至多个队列消息发布到交换器时，消息将拥有一个路由键（routing key），在消息创建时设定。通过队列绑定键，可以把队列绑定到交换器上。消息到达交换器后，RabbitMQ 会将消息的路由键与队列的绑定键进行匹配（针对不同的交换器有不同的路由规则）；

常用的交换器主要分为一下三种(广播模式)：

fanout：如果交换器收到消息，将会广播到所有绑定的队列上

direct：如果路由键完全匹配，消息就被投递到相应的队列

topic：可以使来自不同源头的消息能够到达同一个队列。 使用 topic 交换器时，可以使用通配符

Rabbitmq 里的交换机类型有哪些,都有什么区别?什么是交换机?

1、什么是 Exchange

在 RabbitMQ 中，生产者发送消息不会直接将消息投递到队列中，而是先将消息投递到交换机中， 在由交换机转发到具体的队列， 队列再将消息以推送或者拉取方式给消费者进行消费.

2、路由键（RoutingKey)

生产者将消息发送给交换机的时候， 会指定 RoutingKey 指定路由规则。

3、绑定键（BindingKey)

通过绑定键将交换机与队列关联起来， 这样 RabbitMQ 就知道如何正确地将消息路由到队列。

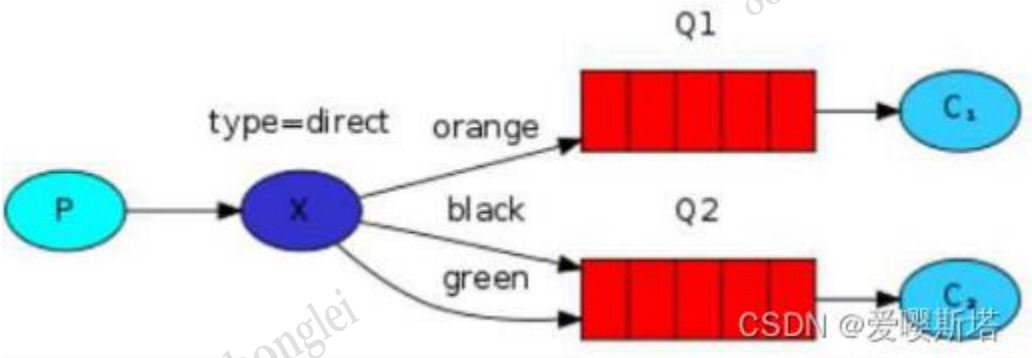
4、关系

生产者将消息发送给哪个 Exchange 是需要由 RoutingKey 决定的，生产者需要将 Exchange 与哪个队列绑定时需要由 BindingKey 决定的。

二、交换机类型和区别

1、直连交换机： Direct exchange

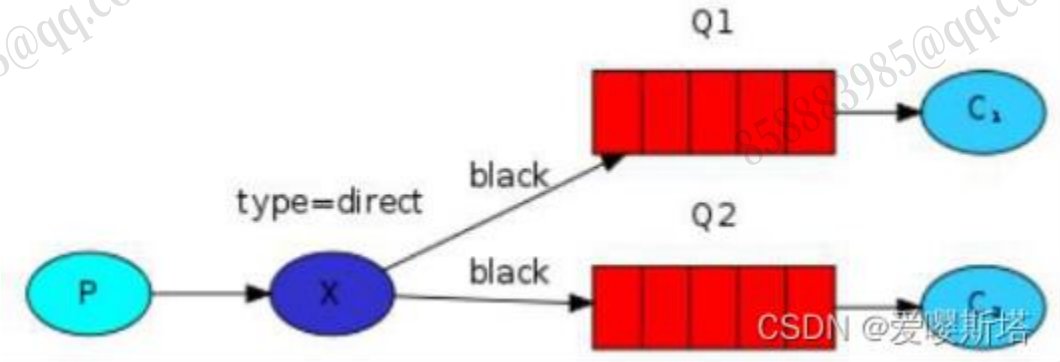
直连交换机的路由算法非常简单： 将消息推送到 binding key 与该消息的 routing key 相同的队列。



直连交换机 X 上绑定了两个队列。第一个队列绑定了绑定 o 键 range, 第二个队列有两个绑定键： black 和 green。

在这种场景下，一 个消息在布时指定了路由键为 orange 将会只被路由到队列 Q1 I 路由键为 black 和 green 的消息都将被路由到队列 Q2。其他的消息都将被丢失。

同一个绑定键可以绑定到不同的队列上去， 可以增加一个交换机 X 与队列 Q2 的绑定键，在这种清况下，直连交换机将会和广播交换机有着相同的行为， 将消息推送到所有匹配的队列。一个路由键为 black 的消息将会同时被推送到队列 Q1 和 Q2。



2、 主题交换机： Topic exchange

直连交换机的缺点：

- 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。

主题交换机的特点：

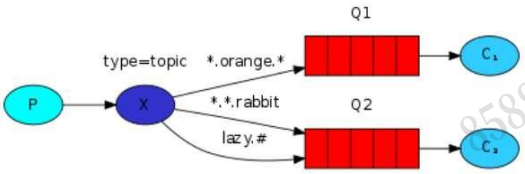
发送到主题交换机的 消息不能有任意的 routing key，必须是由点号分开的一串单词，这些单词可以是任意的，但通常是与消息相关的一些特征。

如以下是几个有效的 routing key:

“stock.usd.nyse”，“nyse.vmw”，“quick.orange.rabb 代”，routing key 的单词可以 有很多，最大限制是 255 bytes。

Topic 交换机的 逻辑与 direct 交换机有点 相似 使用特定路由键发送的消息 将被发送到所有使用匹配绑定键绑定的队列， 然而， 绑定键有两个特殊的情况：

- *表示匹配任意一个单词
- #表示匹配任意一个或多个单词



如：

- routing key quick.orange.rabbit-> queue Ql, Q2
- routing key lazy.orange.elephant-> queue Ql,Q2

延申：当一个队列的绑定键是＂＃＂，它将会接收所有的消息，而不再考虑所接收消息的路由键。

当一个队列的绑定键没有用到"#"和’ "时，它又像 direct 交换一样工作。

2、扇形交换机： Fanout exchange

扇形交换机是最基本的交换机类型，它所能做的事清非常简单广播消息。

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。

3、首部交换机： Headers exchange

类似主题交换机，但是头交换机使用多个消息属性来代替路由键建立路由规则。通过判断消息头的值能否与指定的绑定相匹配来确立路由规则。

此交换机有个重要参数：＂ x-match”

当”x-match”为“any”时，消息头的任意一个值被匹配就可以满足条件

当”x-match”设置为“all”的时候，就需要消息头的所有值都匹配成功

4、默认交换机

实际上是一个由 RabbitMQ 预先声明好的名字为空字符串的直连交换机 (direct exchange) 。

它有一个特殊的属性使得它对于简单应用特别有用处： 那就是每个新建队列 (queue)

都会自动绑定到默认交换机上，绑定的 路由键(routing key) 名称与队列名称相同。

当你声明了一个名为“hello”的队列，RabbitMQ 会自动将其绑定到默认交换机上，绑定（binding）的路由键名称也是为“hello”。

当携带着名为“hello”的路由键的信息被发送到默认交换机的时候，此消息会被默认交换机路由至名为“hello”的队列中

类似 amq.*的名称的交换机：这些是 RabbitMQ 默认创建的交换机。

这些队列名称被预留做 RabbitMQ 内部使用，不能被应用使用，否则抛出 403 错误

5、Dead Letter Exchange(死信交换机)

演示链接：https://blog.csdn.net/weixin_60389087/article/details/123167193

RabbitMQ 作为一个高级消息中间件，提出了死信交换器的概念。

这种交互器专门处理死了的信息（被拒绝可以重新投递的信息不能算死的）。

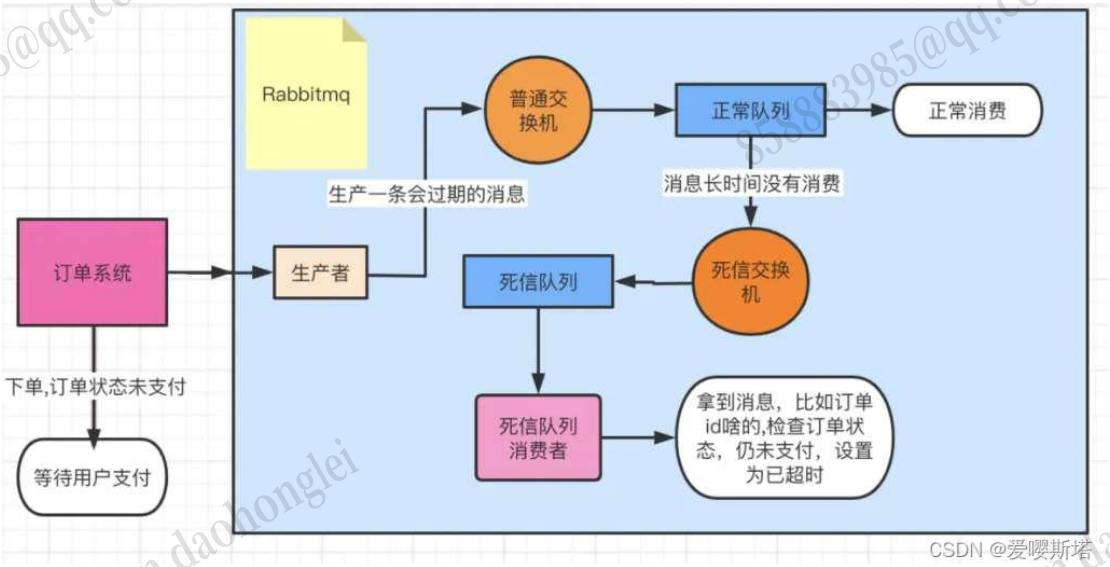
消息变成死信一般是以下三种情况：

1. 消息被拒绝，并且设置 requeue 参数为 false。
2. 消息过期（默认情况下 Rabbit 中的消息不过期，但是可以设置队列的过期时间和信息的过期的效果）
3. 队列达到最大长度（一般当设置了最大队列长度或大小并达到最大值时）

当满足上面三种情况时，消息会变成死信消息，并通过死信交换机投递到相应的队列中。

我们只需要监听相应队列，就可以对死信消息进行最后的处理。

订单超时处理：



生产者生产一条 1 分钟后超时的订单信息到正常交换机 exchange-a 中，消息匹配到队列 queue-a，但一分钟后仍未消费。

消息会被投递到死信交换机 dlx-exchange 中，并发送到私信队列中。

死信队列 dlx-queue 的消费者拿到信息后，根据消息去查询订单的状态，如果仍然是未支付状态，将订单状态更新为超时状态。

6.交换机的属性

Name:交换机名称

Type:交换机类型，direct，topic，fanout，headers

Durability:是否需要持久化，如果持久性，则 RabbitMQ 重启后，交换机还存在

Auto Delete:当最后一个绑定到 Exchange 上的队列删除后，自动删除该 Exchange

Internal:当前 Exchange 是否用于 RabbitMQ 内部使用，默认为 false。

Arguments:扩展参数，用于扩展 AMQP 协议定制使用

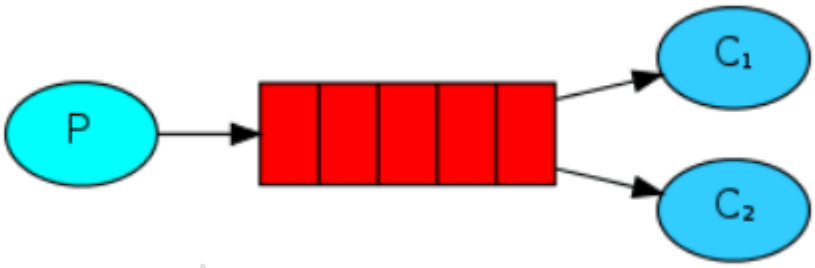
消息传输的模式有哪些?

1.简单模式

简单模式是最简单的消息模式，它包含一个生产者、一个消费者和一个队列。生产者向队列里发送消息，消费者从队列中获取消息并消费。

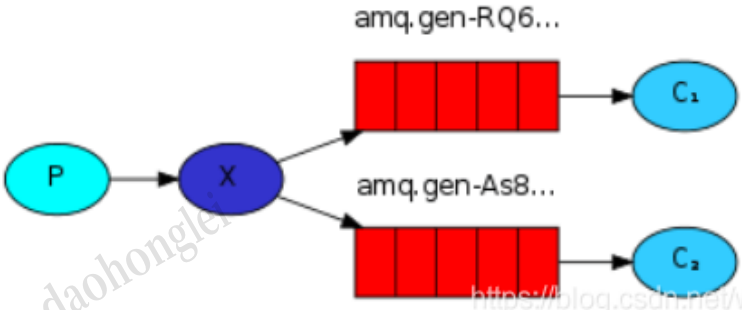
2.工作模式

工作模式是指向多个互相竞争的消费者发送消息的模式，它包含一个生产者、两个消费者和一个队列。两个消费者同时绑定到一个队列上去，当消费者获取消息处理耗时任务时，空闲的消费者从队列中获取并消费消息。



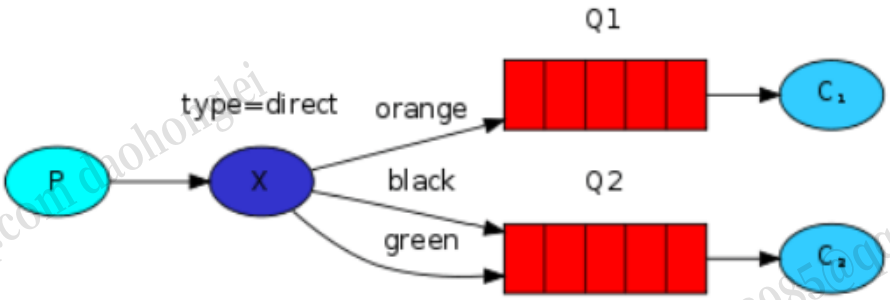
3.发布/订阅模式

发布/订阅模式是指同时向多个消费者消费消息的模式（类似广播的形式），它包含一个生产者、两个消费者、两个队列和一个交换机。两个消费者同时绑定到不同的队列上去，两个队列绑定到交换机上去，生产者通过发送消息到交换机，所有消费者接收并消费消息。



4.路由模式

路由模式是可以根据路由键选择性给多个消费者发送消息的模式，它包含一个生产者、两个消费者、两个队列和一个交换机。两个消费者同时绑定到不同的队列上去，两个队列通过绑定键绑定到交换机上去，生产者发送消息到交换机，交换机通过路由键转发到不同队列，队列绑定的消费者接收并消费消息



(要求路由键和绑定键相同)

常用的交换器主要分为一下三种：

fanout：如果交换器收到消息，将会广播到所有绑定的队列上

direct：如果路由键完全匹配，消息就被投递到相应的队列

topic：可以使来自不同源头的消息能够到达同一个队列。 使用 topic 交换器时，可以使用通配符

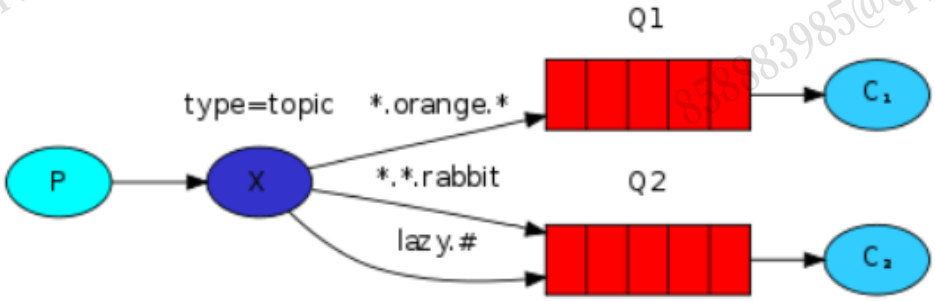
5. 通配符模式:就是绑定键是某种泛泛的符号规则,如果传过来的消息路由键与绑定键匹配,就能传递到对应的消息队列上

特殊匹配符号

*：只能匹配一个单词；

#：可以匹配零个或多个单词。

模式示意图

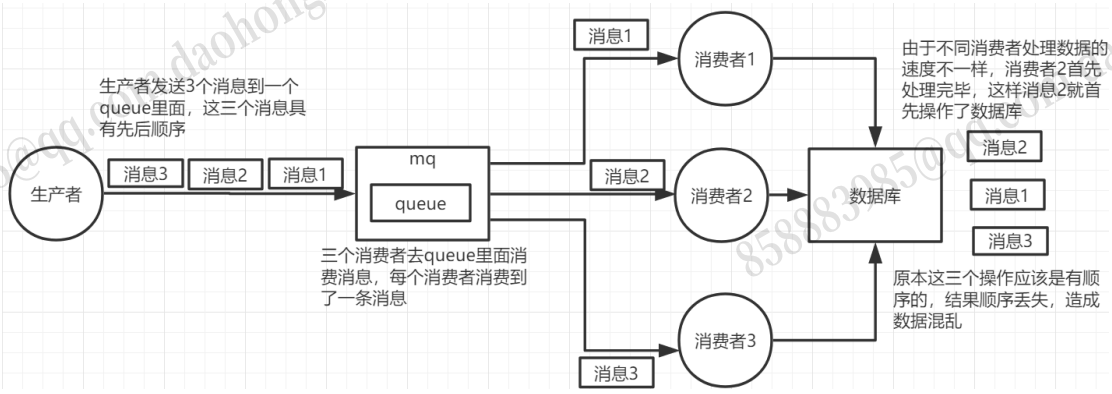


如何保证消息的顺序消费

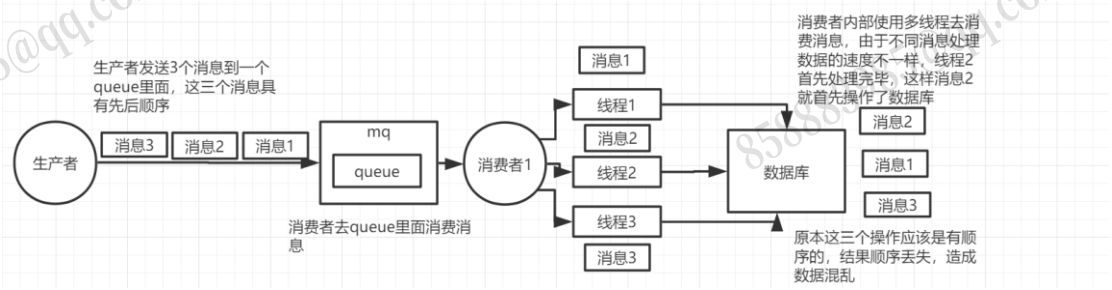
消息在投入到 queue 的时候是有顺序，如果只是单个消费者来处理对应的单个 queue，是不会出现消息错乱的问题。但是在消费的时候有可能多个消费者消费同一个 queue，由于各个消费者处理消息的时间不同，导致消息未能按照预期的顺序处理。其实根本的问题就是如何保证消息按照预期的顺序处理完成。

出现消费顺序错乱的情况

为了提高处理效率，一个 queue 存在多个 consumer

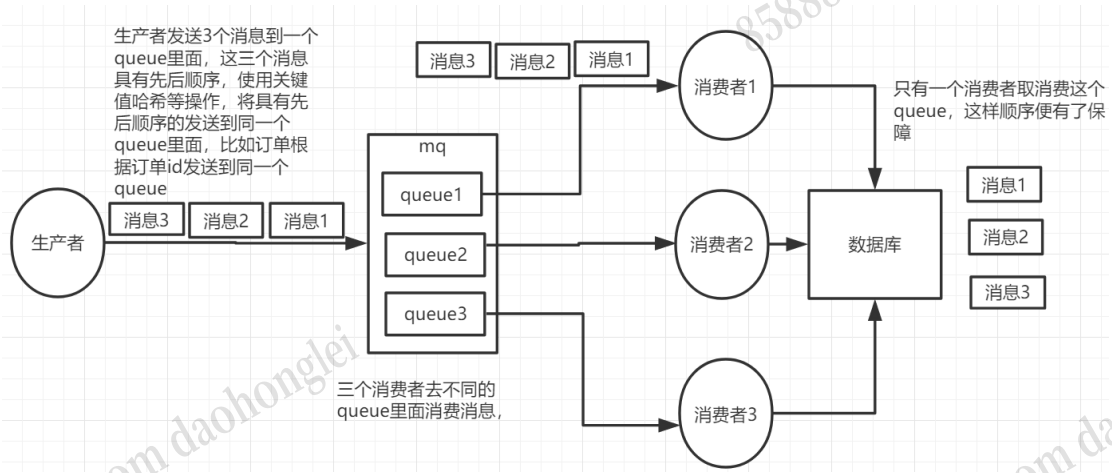


一个 queue 只存在一个 consumer，但是为了提高处理效率，consumer 中使用了多线程进行处理



保证消息顺序性的方法

将原来的一个 queue 拆分成多个 queue，每个 queue 都有一个自己的 consumer。该种方案的核心是生产者在投递消息的时候根据业务数据关键值（例如订单 ID 哈希值对订单队列数取模）来将需要保证先后顺序的同一类数据（同一个订单的数据） 发送到同一个 queue 当中。



一个 queue 就一个 consumer，在 consumer 中维护多个内存队列，根据业务数据关键值（例如订单 ID 哈希值对内存队列数取模）将消息加入到不同的内存队列中，然后多个真正负责处理消息的线程去各自对应的内存队列当中获取消息进行消费。

RabbitMQ 保证消息顺序性总结：

核心思路就是根据业务数据关键值划分成多个消息集合，而且每个消息集合中的消息数据都是有序的，每个消息集合有自己独立的一个 consumer。多个消息集合的存在保证了消息消费的效率，每个有序的消息集合对应单个的 consumer 也保证了消息消费时的有序性。

消息队列如何保证消息的可靠性传输

消息的可靠性传输分为两个问题,一个是保证消息不被重复消费,另一个是保证消息不丢失.

保证消息不重复被消费,就是保证消息的幂等性问题,消息的幂等性是指一个操作执行任意多次所产生的影响均与一次执行的影响相同,在 mq 里,也就是消息只能被消费一次,不能被重复消费.

1. 来看看消息丢失的场景:发送方丢失,可能发送方在发送消息的过程中,出现网络问题等导致 mq 接收不到消息,导致了消息丢失.要解决这个问题,首先可以采用事务机制,在发送消息的时候实现事务机制,若是出现发送失败的情况,可以进行回滚,而让消息重新被发送.但是开启了事务,发送方就必须同步等待事务执行完毕或者回滚,导致消息一多,性能会下降.但是,还有一个更好的办法:可以采用确认机制,发送方在发送消息的时候必须要保证要收到一个确认消息,如果没有收到或者收到失败的确认消息,就说明消息发送失败,要重新进行发送,确认机制是可以采用异步进行的,这样就极大地保证了在保留效率的基础上又能保证消息的不丢失问题.

2. 第二个丢失问题可能是在 mq 方发生的,如果 mq 没有进行持久化,出现了宕机关机等情况,消息就会丢失,解决办法无非就是将消息进行持久化,这样在出现问题的时候可以及时对消息进行恢复.

3. 第三个丢失问题可能在消费方发生,这和发送方丢失问题类似,解决这个问题也是采用确认机制,这样一来就可以实现效率上的保证和消息不丢失的保证.

但是解决了这些问题,就会产生下面的幂等性问题:

我们都知道 mq 是可以进行重发的,且只有在它认为失败的情况会进行重发.什么时候 mq 会认为它发送给消费者的消息是失败的呢?也就是超出了它等待消费者响应的时间,这是一个超时时间,若是过了这个时间消费者仍然没有响应,说明 mq 发送失败,就会进行重试,而其

实这个时候消费者可能是没有失败的,它只是因为某个原因导致消费超出了 mq 的等待时间而已,这个时候 mq 再发送一次消息,消费者就会重复消费.

4. 实现幂等性消费:

   a) 通过数据库：比如处理订单时，记录订单 ID，在消费前，去数据库中进行查询该记录是否存在，如果存在则直接返回。

   b) 使用全局唯一 ID，再配合第三组主键做消费记录，比如使用 redis 的 set 结构，生产者发送消息时给消息分配一个全局 ID，在每次消费者开始消费前，先去 redis 中查询有没有消费记录，如果消费过则不进行处理，如果没消费过，则进行处理，消费完之后，就将这个 ID 以 k-v 的形式存入 redis 中(过期时间根据具体情况设置)。

如何处理消息堆积情况?

场景题：几千万条数据在 MQ 里积压了七八个小时。

1. **出现该问题的原因：**

消息堆积往往是生产者的生产速度与消费者的消费速度不匹配导致的。有可能就是消费者消费能力弱，渐渐地消息就积压了，也有可能是因为消息消费失败反复复重试造成的，也有可能是消费端出了问题，导致不消费了或者消费极其慢。比如，消费端每次消费之后要写 mysql，结果 mysql 挂了，消费端 hang 住了不动了，或者消费者本地依赖的一个东西挂了，导致消费者挂了。

所以如果是 bug 则处理 bug；如果是因为本身消费能力较弱，则优化消费逻辑，比如优化前是一条一条消息消费处理的，那么就可以批量处理进行优化。

2. **临时扩容，快速处理积压的消息：**

a) 先修复 consumer 的问题，确保其恢复消费速度，然后将现有的 consumer 都停掉；

b) 临时创建原先 N 倍数量的 queue ，然后写一个临时分发数据的消费者程序，将该程序部署上去消费队列中积压的数据，消费之后不做任何耗时处理，直接均匀轮询写入临时建立好的 N 倍数量的 queue 中；

c) 接着，临时征用 N 倍的机器来部署 consumer，每个 consumer 消费一个临时 queue 的数据

d) 等快速消费完积压数据之后，恢复原先部署架构 ，重新用原先的 consumer 机器消费消息。

这种做法相当于临时将 queue 资源和 consumer 资源扩大 N 倍，以正常 N 倍速度消费。

3. **MQ 长时间未处理导致 MQ 写满的情况如何处理：**

如果消息积压在 MQ 里，并且长时间都没处理掉，导致 MQ 都快写满了，这种情况肯定是临时扩容方案执行太慢，这种时候只好采用 “丢弃+批量重导” 的方式来解决了。首先，临时写个程序，连接到 mq 里面消费数据，消费一个丢弃一个，快速消费掉积压的消息，降低 MQ 的压力，然后在流量低峰期时去手动查询重导丢失的这部分数据

4. **短时间内无法扩容或者扩容无法完全解决问题**

可以尝试降低一些非核心业务的消息处理,其次可以通过监控排查,优化消费者端的业务代码或者查看是否存在一些消息被重复消息的情况.

如何保证消息队列的高可用?

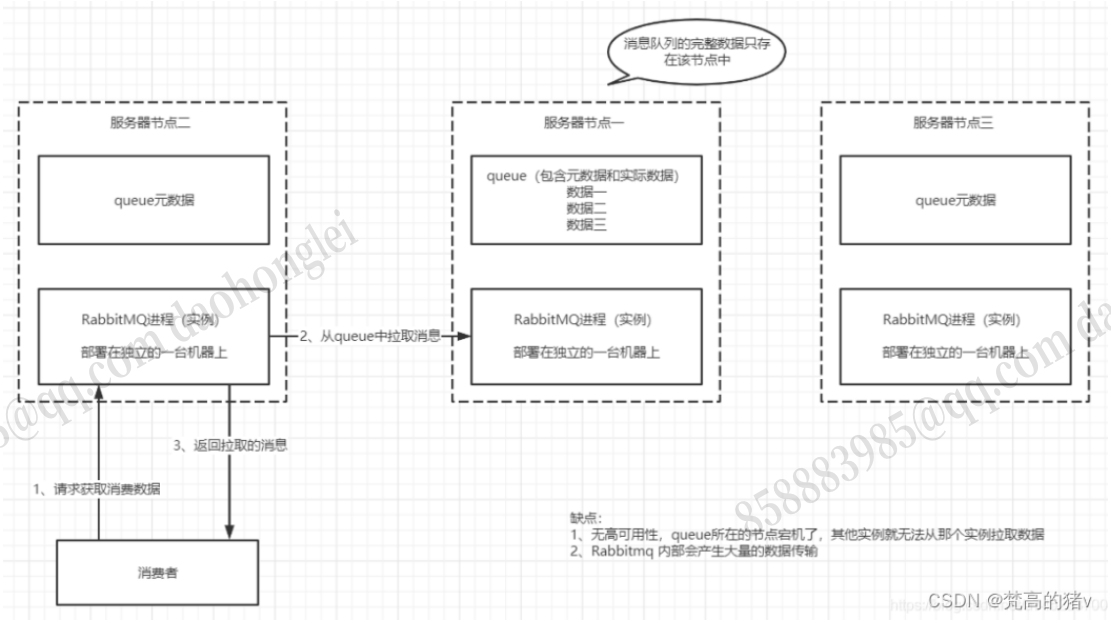
RabbitMQ 是基于主从（非分布式）做高可用性的，RabbitMQ 有三种模式：单机模式、普通集群模式、镜像集群模式

1. **单机模式：一般没人生产用单机模式**

2. **普通集群模式：**

普通集群模式用于提高系统的吞吐量，通过添加节点来线性扩展消息队列的吞吐量。也就是在多台机器上启动多个 RabbitMQ 实例，而队列 queue 的消息只会存放在其中一个 RabbitMQ 实例上，但是每个实例都同步 queue 的元数据（元数据是 queue 的一些配置信息，通过元数据，可以找到 queue 所在实例）。消费的时候，如果连接到了另外的实例，那么该实例就会从数据实际所在的实例上的 queue 拉取消息过来，就是说让集群中多个节点来服务某个 queue 的读写操作

但普通集群模式的缺点在于：无高可用性，queue 所在的节点宕机了，其他实例就无法从那个实例拉取数据；RabbitMQ 内部也会产生大量的数据传输。



3. **镜像队列集群模式：**

镜像队列集群是 RabbitMQ 真正的高可用模式，集群中一般会包含一个主节点 master 和若干个从节点 slave，如果 master 由于某种原因失效，那么按照 slave 加入的时间排序，"资历最老"的 slave 会被提升为新的 master。

镜像队列下，所有的消息只会向 master 发送，再由 master 将命令的执行结果广播给

slave，所以 master 与 slave 节点的状态是相同的。比如，每次写消息到 queue 时，master 会自动将消息同步到各个 slave 实例的 queue；如果消费者与 slave 建立连接并进行订阅消费，其实质上也是从 master 上获取消息，只不过看似是从 slave 上消费而已，比如消费者与 slave 建立了 TCP 连接并执行 Basic.Get 的操作，那么也是由 slave 将 Basic.Get 请求发往 master，再由 master 准备好数据返回给 slave，最后由 slave 投递给消费者。

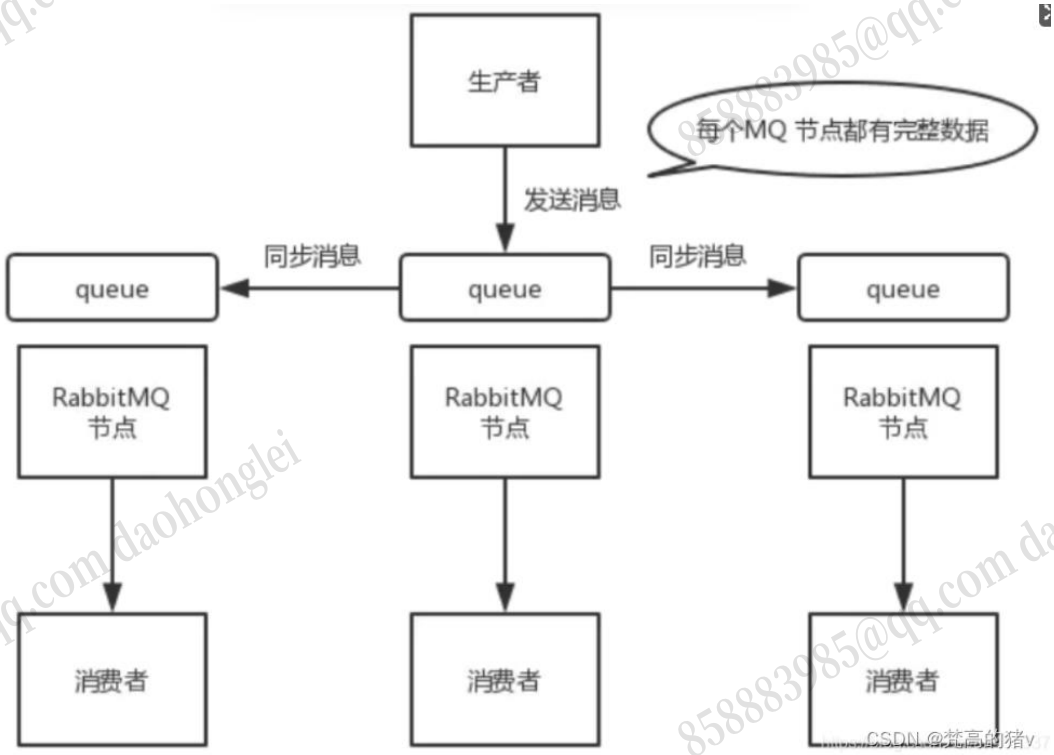
从上面可以看出，队列的元数据和消息会存在于多个实例上，也就是说每个 RabbitMQ 节点都有这个 queue 的完整镜像，任何一个机器宕机了，其它机器节点还包含了这个 queue 的完整数据，其他消费者都可以到其它节点上去消费数据。

缺点：

1. 性能开销大，消息需要同步到所有机器上，导致网络带宽压力和消耗很重
2. 非分布式，没有扩展性，如果 queue 的数据量大到这个机器上的容量无法容纳了，此时该方案就会出现问题了

如何开启镜像集群模式呢?

在 RabbitMQ 的管理控制台 Admin 页面下，新增一个镜像集群模式的策略，指定的时候是可以要求数据同步到所有节点的，也可以要求同步到指定数量的节点，再次创建 queue 的时候，应用这个策略，就会自动将数据同步到其他的节点上去了。



rabbitmq 持久化机制

RabbitMQ 的消息默认存放在内存上面，如果不特别声明设置，消息不会持久化保存到硬盘上面的，如果节点重启或者意外 crash 掉，消息就会丢失。所以就要对消息进行持久化处理。

rabbitmq 的持久化分为队列持久化、消息持久化和交换器持久化。

1. 队列的持久化是在定义队列时的 durable 参数来决定的，当 durable 为 true 时，才代表队列会持久化。

```
Connection connection = connectionFactory.newConnection();
Channel channel = connection.createChannel();
//第二个餐胡设置为 true，代表队列持久化
channel.queueDeclare("queue.persistent.name", true, false, false, null);
```

2. 如果要在重启后保持消息的持久化必须设置消息是持久化的标识。

```
//通过传入 MessageProperties.PERSISTENT_PLAIN 就可以实现消息持久化
channel.basicPublish("exchange.persistent", "persistent",
MessageProperties.PERSISTENT_TEXT_PLAIN,
"persistent_test_message".getBytes());
```

3. 上面阐述了队列的持久化和消息的持久化，如果不设置 exchange 的持久化对消息的可靠性来说没有什么影响，**但是同样如果 exchange 不设置持久化，那么当 broker 服务重启之后，exchange 将不复存在，那么既而发送方 rabbitmq producer 就无法正常发送消息。**这里建议，同样设置 exchange 的持久化。exchange 的持久化设置也特别简单.

```
//durable 为 true 则开启持久化
Exchange.DeclareOk exchangeDeclare(String exchange,String type,boolean
durable)throws IOException
```

所以一般三个持久化设置都要进行.

可以对所有消息都持久化吗?

不可以

持久化的操作是将数据写入磁盘中,效率上肯定比写入内存中要慢很多倍.而我们一般用 mq 会处理很多业务消息,若是所有消息都持久化,压力无疑是巨大的.所以持久化策略需要综合考虑,以及可能遇到的问题和解决方案,或者我们可以让一些必要数据持久化.

Rabbitmq 事务机制

1. 概述

在使用 RabbitMQ 的时候，我们可以通过消息持久化操作来解决因为服务器的异常奔溃导致的消息丢失，除此之外我们还会遇到一个问题，当消息的发布者在将消息发送出去之后，消息到底有没有正确到达 broker 代理服务器呢？如果不进行特殊配置的话，默认情况下发布操作是不会返回任何信息给生产者的，也就是默认情况下我们的生产者是不知道消息有没有正确到达 broker 的，如果在消息到达 broker 之前已经丢失的话，持久化操作也解决不了这个问题，因为消息根本就没到达代理服务器，你怎么进行持久化，那么这个问题该怎么解决呢?

RabbitMQ 为我们提供了两种方式：

- 通过 AMQP 事务机制实现，这也是 AMQP 协议层面提供的解决方案；
- 通过将 channel 设置成 confirm 模式来实现；

2. 事务机制

这里首先探讨下 RabbitMQ 事务机制。

RabbitMQ 中与事务机制有关的方法有三个：txSelect(), txCommit()以及 txRollback(), txSelect 用于将当前 channel 设置成 transaction 模式，txCommit 用于提交事务，txRollback 用于回滚事务，在通过 txSelect 开启事务之后，我们便可以发布消息给 broker 代理服务器了，如果 txCommit 提交成功了，则消息一定到达了 broker 了，如果在 txCommit 执行之前 broker 异常崩溃或者由于其他原因抛出异常，这个时候我们便可以捕获异常通过 txRollback 回滚事务了。

```
public class P1 {


    private static final String QUEUE_NAME = "test_tx";
    public static void main(String[] args) throws IOException, TimeoutException {
```

```
        Connection connection = ConnectionUtils.getConnection();
        Channel channel = connection.createChannel();


        channel.queueDeclare(QUEUE_NAME,false,false,true,null);
        SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-
dd 'at' HH:mm:ss z");
        Date date = new Date(System.currentTimeMillis());
        String message = simpleDateFormat.format(date);


        try {
            channel.txSelect();//开始事务
            channel.basicPublish("",QUEUE_NAME,null,message.getBytes());
            channel.txCommit();//提交事务
        }catch (Exception e){
            channel.txRollback();//回滚事务
            System.out.println("send message txRollback");
        }
        channel.close();
        connection.close();
    }
}
```

谈谈你对死信队列的理解

当 queue 消息队列中的消息由于一些原因没办法被消费，如果这些消息一直没办法被处理，就会从这个正常的消息队列转移到死信消息队列中。

应用场景：当用户下单之后，如果一直不支付，那么用户下单的这条消息就会被存放到死信队列中去。

死信的来源

1. 当 queue 消息队列满的时候，再进来的消息就会被放到死信队列中去。
2. 在消息应答的时候，如果消费者一直没有告诉 RabbitMQ 有没有成功处理消息，那么 RabbitMQ 消息队列就不清楚自己到底要不要删除这条消息，这个时候消息队列中的消息一直没办法处理，这样这条消息也会被放到死信队列中去。
3. 消息 TTL 过期，什么意思？TTL 的全拼是 Time To Live 意思是指存活时间，就是消息队列中存放的消息一般都是有一定时间的，超过了这个时间，这条消息就会被放到死信队列中去。

如何去实现死信队列?

生产者:

发送消息.

普通消费者:

1. 在用 Map 声明普通队列参数的时候,可以先设置死信队列的路由 key 和交换机,让普通该队列处理不来的消息转到对应的死信队列上.
2. 然后声明普通队列,普通交换机,死信队列,死信交换机.
3. 之后消费普通队列消息

```
public class Consumer01 {
```

```
    //普通交换机的名称
    public static final String NORMAL_EXCHANGE="normal_exchange";
    //死信交换机的名称
    public static final String DEAD_EXCHANGE="dead_exchange";
    //普通队列的名称
    public static final String NORMAL_QUEUE="normal_queue";
    //死信队列的名称
    public static final String DEAD_QUEUE="dead_queue";

    public static void main(String[] args) throws IOException, TimeoutException {
        Channel channel = RabbitMQUtil.getChannel();

        //声明死信和普通交换机类型为 direct 类型
        channel.exchangeDeclare(NORMAL_EXCHANGE,
BuiltinExchangeType.DIRECT);
        channel.exchangeDeclare(DEAD_EXCHANGE,BuiltinExchangeType.DIRECT);

        //声明普通队列
        Map<String,Object> arguments=new HashMap<>();
        arguments.put("x-dead-letter-exchange",DEAD_EXCHANGE);
        arguments.put("x-dead-letter-routing-key","lisi");
        channel.queueDeclare(NORMAL_QUEUE,false,false,false,arguments);
```

```
        //声明死信队列
        channel.queueDeclare(DEAD_QUEUE,false,false,false,null);

        //绑定普通的交换机与普通的队列
        channel.queueBind(NORMAL_QUEUE,NORMAL_EXCHANGE,"zhangsan");

        //绑定死信的交换机与死信的队列
        channel.queueBind(DEAD_QUEUE,DEAD_EXCHANGE,"lisi");

        //如果能成功接收到消息会调用的回调函数
        DeliverCallback deliverCallback=(consumerTag, message)->{
            System.out.println("Consumer01 接收者接收到的消息:"+new
String(message.getBody()));
        };

        //如果取消从消息队列中获取消息时会调用的回调函数
        CancelCallback cancelCallback= consumerTag->{
            System.out.println("消息消费被中断");
        };

        channel.basicConsume(NORMAL_QUEUE,true,deliverCallback,cancelCallback);
```

```
    }
}
```

死信消费者:

1)消费死信队列即可,消费过程和普通队列一样.

SpringBoot 如何整合 Rabbitmq

这种低级的题目在工作了几年的程序员估计闻不到,但是想我们这种刚出来的小白估计会问到

1. 导入 spring-boot-starter-amqp 的包
2. 编写 config 配置类,配置相关交换机,队列,绑定键
3. 利用@RabbitListener(要绑定的队列)+ @RabbitHandler 比如前一个注解标注在类上,后一个注解标注在方法上,用来监听某一个队列后,一旦队列有消息就做出相应操作
4. 注入 RabbitmqTemplate 组件,然后利用相应的 api 进行发送消息,比如 convertAndSend 方法
5. 在处理订单超时这种情况便是如上方案: 先配置好队列的过期时间,一旦生成订单的消息在消息队列里没有被消费,那么这个时候就会被放入到死信队列里面,这个时候,我们就可以监听这个死信队列,一旦死信队列里面产生了消息,就执行 dao 的方法将相应的订单消息删除.
6. 其次,如果你想保证消息的可靠性传输,可以设置应答机制和返回机制,利用 rabbtemplate 里的 setConfirmCallback 和 setReturnCallback 方法,前者是在消息成功到达之后会返回一个回调,后者是消息没有成功到达会返回一个回调,最后利用

@PostConstruct 让这两个机制发生在容器初始化的时候被调用就可以了.

https://github.com/daohonglei/javaStereotypedWriting

https://gitee.com/daohonglei/javaStereotypedWriting