
回到首页

芋道源码 —— 知识星球

我是一段不羁的公告！
记得给芋艿这 3 个项目加油，添加一个 STAR 噢。
https://github.com/YunaiV/SpringBoot-Labs
https://github.com/YunaiV/onemall
https://github.com/YunaiV/ruoyi-vue-pro
2019-11-13

精尽 RabbitMQ 面试题「最新更新时间：2023-07」

以下面试题，基于网络整理，和自己编辑。具体参考的文章，会在文末给出所有的链接。

如果胖友有自己的疑问，欢迎在星球提问，我们一起整理吊吊的 RabbitMQ 面试题的大保健。

而题目的难度，芋艿尽量按照从容易到困难的顺序，逐步下去。

> 芋艿：我们生产主要使用的是 RocketMQ ，所以对 RabbitMQ 灰常不熟悉（ 看了《RabbitMQ 实战指南》入门级的萌新）。所以本文以整理为主。如果胖友在实际面试场景下，碰到 RabbitMQ 的一些问题，可以返回到星球，芋艿可以去请教厮哒哒，然后来装比。

RabbitMQ 是什么？

RabbitMQ 是一个由 Erlang 语言开发的 AMQP 的开源实现。

> 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。

RabbitMQ 最初起源于金融系统，用于在分布式系统中存储转发消息，在易用性、扩展性、高可用性等方面表现不俗。具体特点包括：

1、可靠性（Reliability）

RabbitMQ 使用一些机制来保证可靠性，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 来路由消息的。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 。

3、消息集群（Clustering）

多个 RabbitMQ 服务器可以组成一个集群，形成一个逻辑 Broker 。

4、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。

5、多种协议（Multi-protocol）

RabbitMQ 支持多种消息队列协议，比如 STOMP、MQTT 等等。

6、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。

7、管理界面（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 Broker 的许多方面。

8、跟踪机制（Tracing）

如果消息异常，RabbitMQ 提供了消息跟踪机制，使用者可以找出发生了什么。

9、插件机制（Plugin System）

RabbitMQ 提供了许多插件，来从多方面进行扩展，也可以编写自己的插件。

更详细的，推荐阅读 《消息队列之 RabbitMQ》 。它提供了：

1、RabbitMQ 的介绍
2、RabbitMQ 的概念
3、RabbitMQ 的 Server 安装
4、RabbitMQ 的 Java Client 使用示例
5、RabbitMQ 的集群

RabbitMQ 中的 Broker 是指什么？Cluster 又是指什么？

Broker ，是指一个或多个 erlang node 的逻辑分组，且 node 上运行着 RabbitMQ 应用程序。
Cluster ，是在 Broker 的基础之上，增加了 node 之间共享元数据的约束。

vhost 是什么？起什么作用？

vhost 可以理解为虚拟 Broker ，即 mini-RabbitMQ server 。其内部均含有独立的 queue、exchange 和 binding 等，但最最重要的是，其拥有独立的权限系统，可以做到 vhost 范围的用户控制。当然，从 RabbitMQ 的全局角度，vhost 可以作为不同权限隔离的手段（一个典型的例子就是不同的应用可以跑在不同的 vhost 中）。

这个，和 Tomcat、Nginx、Apache 的 vhost 是一样的概念。

什么是元数据？元数据分为哪些类型？包括哪些内容？

在非 Cluster 模式下，元数据主要分为：

Queue 元数据（queue 名字和属性等）
Exchange 元数据（exchange 名字、类型和属性等）
Binding 元数据（存放路由关系的查找表）

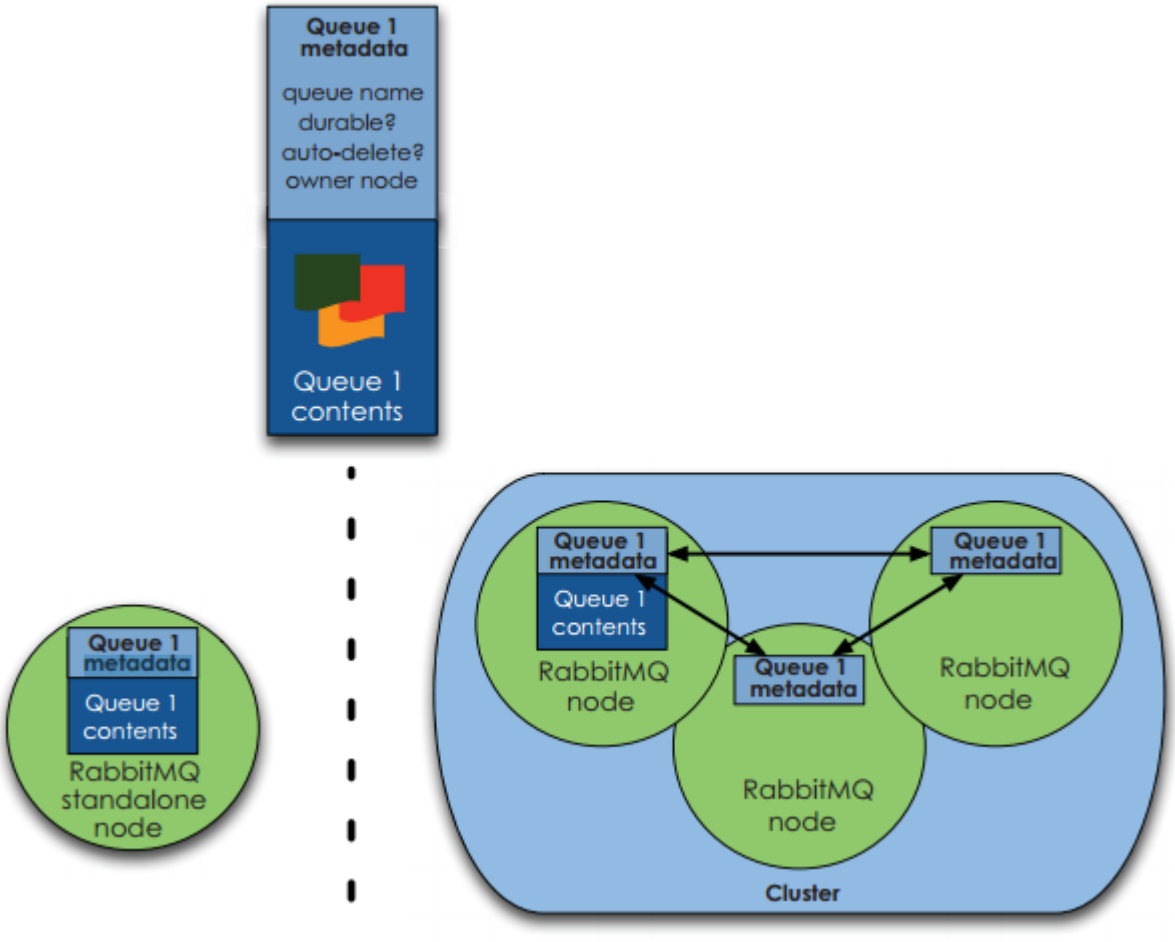
Vhost 元数据（vhost 范围内针对前三者的名字空间约束和安全属性设置）。

与 Cluster 相关的元数据有哪些？元数据是如何保存的？元数据在 Cluster 中是如何分布的？

在 Cluster 模式下，还包括 Cluster 中 node 位置信息和 node 关系信息。

元数据根据 erlang node 的类型确定是仅保存于 RAM 中，还是同时保存在 RAM 和 disk 上。元数据在 Cluster 中是全 node 分布的。

下图所示为 queue 的元数据在单 node 和 cluster 两种模式下的分布图：



RAM node 和 Disk node 的区别？

RAM node 仅将 fabric（即 queue、exchange 和 binding等 RabbitMQ基础构件）相关元数据保存到内存中，但 Disk node 会在内存和磁盘中均进行存储。
RAM node 上唯一会存储到磁盘上的元数据是 Cluster 中使用的 Disk node 的地址。并且要求在 RabbitMQ Cluster 中至少存在一个 Disk node 。

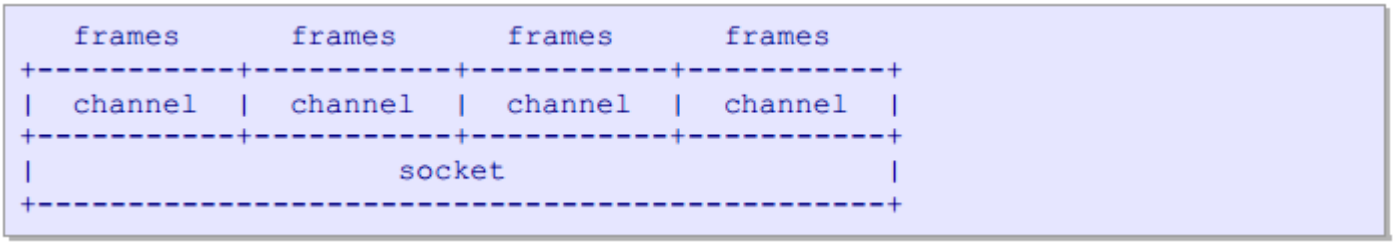
RabbitMQ 概念里的 channel、exchange 和 queue 是什么？

queue 具有自己的 erlang 进程；
exchange 内部实现为保存 binding 关系的查找表；
channel 是实际进行路由工作的实体，即负责按照 routing_key 将 message 投递给 queue 。

由 AMQP 协议描述可知，channel 是真实 TCP 连接之上的虚拟连接，所有 AMQP 命令都是通过

channel 发送的，且每一个 channel 有唯一的 ID 。

一个 channel 只能被单独一个操作系统线程使用，故投递到特定 channel 上的 message 是有顺序的。但一个操作系统线程上允许使用多个 channel 。
channel 号为 0 的 channel 用于处理所有对于当前 connection 全局有效的帧，而 1-65535 号 channel 用于处理和特定 channel 相关的帧。
AMQP 协议给出的 channel 复用模型如下：



- 其中每一个 channel 运行在一个独立的线程上，多线程共享同一个 socket 。

消息基于什么传输？

由于 TCP 连接的创建和销毁开销较大，且并发数受系统资源限制，会造成性能瓶颈。RabbitMQ 使用信道的方式来传输数据。信道是建立在真实的 TCP 连接内的虚拟连接，且每条 TCP连接上的信道数量没有限制。

RabbitMQ 上的一个 queue 中存放的 message 是否有数量限制？

可以认为是无限制，因为限制取决于机器的内存，但是消息过多会导致处理效率的下降。

下面的几个问题，Cluster 相关。

在单 node 系统和多 node 构成的 cluster 系统中声明 queue、exchange ，以及进行 binding 会有什么不同？

当你在单 node 上声明 queue 时，只要该 node 上相关元数据进行了变更，你就会得到 Queue.Declare-ok 回应；而在 cluster 上声明 queue ，则要求 cluster 上的全部 node 都要进行元数据成功更新，才会得到 Queue.Declare-ok 回应。
另外，若 node 类型为 RAM node 则变更的数据仅保存在内存中，若类型为 Disk node 则还要变更保存在磁盘上的数据。

客户端连接到 Cluster 中的任意 node 上是否都能正常工作？

是的。客户端感觉不到有何不同。

若 Cluster 中拥有某个 queue 的 owner node 失效了，且该 queue 被声明具有 durable 属性，是否能够成功从其他 node 上重新声明该 queue ？

不能，在这种情况下，将得到 404 NOT_FOUND 错误。只能等 queue 所属的 node 恢复后才能使用该 queue 。
但若该 queue 本身不具有 durable 属性，则可在其他 node 上重新声明。

Cluster 中 node 的失效会对 consumer 产生什么影响？若是在 cluster 中创建了 mirrored queue ，这时 node 失效会对 consumer 产生什么影响？

若是 consumer 所连接的那个 node 失效（无论该 node 是否为 consumer 所订阅 queue 的 owner node），则 consumer 会在发现 TCP 连接断开时，按标准行为执行重连逻辑，并根据“Assume Nothing” 原则重建相应的 fabric 即可。
若是失效的 node 为 consumer 订阅 queue 的 owner node，则 consumer 只能通过

Consumer Cancellation Notification 机制来检测与该 queue 订阅关系的终止，否则会出现傻等却没有任何消息来到的问题。

Consumer Cancellation Notification 机制用于什么场景？

用于保证当镜像 queue 中 master 挂掉时，连接到 slave 上的 consumer 可以收到自身 consume 被取消的通知，进而可以重新执行 consume 动作从新选出的 master 出获得消息。

若不采用该机制，连接到 slave 上的 consumer 将不会感知 master 挂掉这个事情，导致后续无法再收到新 master 广播出来的 message 。

另外，因为在镜像 queue 模式下，存在将 message 进行 requeue 的可能，所以实现 consumer 的逻辑时需要能够正确处理出现重复 message 的情况。

能够在地理上分开的不同数据中心使用 RabbitMQ cluster 么？

不能。

第一，你无法控制所创建的 queue 实际分布在 cluster 里的哪个 node 上（一般使用 HAProxy + cluster 模型时都是这样），这可能会导致各种跨地域访问时的常见问题。
第二，Erlang 的 OTP 通信框架对延迟的容忍度有限，这可能会触发各种超时，导致业务疲于处理。
第三，在广域网上的连接失效问题将导致经典的“脑裂”问题，而 RabbitMQ 目前无法处理。（该问题主要是说 Mnesia）

如何确保消息正确地发送至 RabbitMQ？

RabbitMQ 使用发送方确认模式，确保消息正确地发送到 RabbitMQ。

发送方确认模式：将信道设置成 confirm 模式（发送方确认模式），则所有在信道上发布的消息都会被指派一个唯一的 ID 。一旦消息被投递到目的队列后，或者消息被写入磁盘后（可持久化的消息），信道会发送一个确认给生产者（包含消息唯一ID）。如果 RabbitMQ 发生内部错误从而导致消息丢失，会发送一条 nack（not acknowledged，未确认）消息。
发送方确认模式是异步的，生产者应用程序在等待确认的同时，可以继续发送消息。当确认消息到达生产者应用程序，生产者应用程序的回调方法就会被触发来处理确认消息。

具体的代码实现，可以看看 《芋道 Spring Boot 消息队列 RabbitMQ 入门》的「14. 生产者的发送确认」 小节

向不存在的 exchange 发 publish 消息会发生什么？向不存在的 queue 执行 consume 动作会发生什么？

都会收到 Channel.Close 信令告之不存在（内含原因 404 NOT_FOUND）。

什么情况下会出现 blackholed 问题？

blackholed ，对应中文为“黑洞”。

blackholed 问题是指，向 exchange 投递了 message ，而由于各种原因导致该 message 丢失，但发送者却不知道。可导致 blackholed 的情况：

1、向未绑定 queue 的 exchange 发送 message 。
2、exchange 以 binding_key key_A 绑定了 queue queue_A，但向该 exchange 发送 message 使用的 routing_key 却是 key_B 。

如何防止出现 blackholed 问题？

没有特别好的办法，只能在具体实践中通过各种方式保证相关 fabric 的存在。另外，如果在执行 Basic.Publish 时设置 mandatory=true ，则在遇到可能出现 blackholed 情况时，服务器会通过返回 Basic.Return 告之当前 message 无法被正确投递（内含原因 312 NO_ROUTE）。

具体的代码实现，可以看看 《芋道 Spring Boot 消息队列 RabbitMQ 入门》的「14.3 ReturnCallback」 小节

routing_key 和 binding_key 的最大长度是多少？

255 字节。

消息怎么路由？

从概念上来说，消息路由必须有三部分：交换器、路由、绑定。

生产者把消息发布到交换器上；

绑定决定了消息如何从路由器路由到特定的队列；

如果一个路由绑定了两个队列，那么发送给该路由时，这两个队列都会增加一条消息。

消息最终到达队列，并被消费者接收。

详细来说，就是：

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。
通过队列路由键，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路由键进行匹配（针对不同的交换器有不同的路由规则）。如果能够匹配到队列，则消息会投递到相应队列中；如果不能匹配到任何队列，消息将进入 “黑洞”。

常用的交换器主要分为一下三种：

direct：如果路由键完全匹配，消息就被投递到相应的队列。
fanout：如果交换器收到消息，将会广播到所有绑定的队列上。
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，比如：“*” 匹配特定位置的任意文本， “.” 把路由键分为了几部分，“#” 匹配所有规则等。特别注意：发往 topic 交换器的消息不能随意的设置选择键（routing_key），必须是由 "." 隔开的一系列的标识符组成。

具体的代码实现，可以看看 《芋道 Spring Boot 消息队列 RabbitMQ 入门》的「3. 快速入门」小节

如何确保消息接收方消费了消息？

RabbitMQ 使用接收方消息确认机制，确保消息接收方消费了消息。

接收方消息确认机制：消费者接收每一条消息后都必须进行确认（消息接收和消息确认是两个不同操作）。只有消费者确认了消息，RabbitMQ 才能安全地把消息从队列中删除。

这里并没有用到超时机制，RabbitMQ 仅通过 Consumer 的连接中断来确认是否需要重新发送消息。也就是说，只要连接不中断，RabbitMQ 给了 Consumer 足够长的时间来处理消息。

下面罗列几种特殊情况：

> 如果消费者接收到消息，在确认之前断开了连接或取消订阅，RabbitMQ 会认为消息没有被分发，然后重新分发给下一个订阅的消费者。（可能存在消息重复消费的隐患，需要根据 bizId 去重）
> 如果消费者接收到消息却没有确认消息，连接也未断开，则 RabbitMQ 认为该消费者繁忙，将不会给该消费者分发更多的消息。

具体的代码实现，可以看看 《芋道 Spring Boot 消息队列 RabbitMQ 入门》的「13. 消费者的消息确认」 小节。

如何避免消息重复投递或重复消费？

在消息生产时，MQ 内部针对每条生产者发送的消息生成一个 inner-msg-id ，作为去重和幂等的依据（消息投递失败并重传），避免重复的消息进入队列

在消息消费时，要求消息体中必须要有一个 bizId（对于同一业务全局唯一，如支付 ID、订单 ID、帖子 ID 等）作为去重和幂等的依据，避免同一条消息被重复消费。

消息如何分发？

若该队列至少有一个消费者订阅，消息将以循环（round-robin）的方式发送给消费者。每条消息只会分发给一个订阅的消费者（前提是消费者能够正常处理消息并进行确认）。

RabbitMQ 有几种消费模式？

RabbitMQ 有 pull 和 push 两种消费模式。具体的使用，可参见 《RabbitMQ 之 Consumer 消费模式（Push & Pull）》 文章。

为什么不应该对所有的 message 都使用持久化机制？

首先，必然导致性能的下降，因为写磁盘比写 RAM 慢的多，message 的吞吐量可能有 10 倍的差距。

其次，message 的持久化机制用在 RabbitMQ 的内置 Cluster 方案时会出现“坑爹”问题。矛盾点在于：

> 若 message 设置了 persistent 属性，但 queue 未设置 durable 属性，那么当该 queue 的 owner node 出现异常后，在未重建该 queue 前，发往该 queue 的 message 将被 blackholed 。
> 若 message 设置了 persistent 属性，同时 queue 也设置了 durable 属性，那么当 queue 的 owner node 异常且无法重启的情况下，则该 queue 无法在其他 node 上重建，只能等待其 owner node 重启后，才能恢复该 queue 的使用，而在这段时间内发送给该 queue 的 message 将被 blackholed 。

所以，是否要对 message 进行持久化，需要综合考虑性能需要，以及可能遇到的问题。若想达到 100,000 条/秒以上的消息吞吐量（单 RabbitMQ 服务器），则要么使用其他的方式来确保 message 的可靠 delivery ，要么使用非常快速的存储系统以支持全持久化（例如使用 SSD）。另外一种处理原则是：仅对关键消息作持久化处理（根据业务重要程度），且应该保证关键消息的量不会导致性能瓶颈。

RabbitMQ 允许发送的 message 最大可达多大？

根据 AMQP 协议规定，消息体的大小由 64-bit 的值来指定，所以你就可以知道到底能发多大的数据了。

为什么说保证 message 被可靠持久化的条件是 queue 和 exchange 具有 durable 属性，同时 message 具有 persistent 属性才行？

binding 关系可以表示为 exchange－binding－queue 。从文档中我们知道，若要求投递的 message 能够不丢失，要求 message 本身设置 persistent 属性，同时要求 exchange 和 queue 都设置 durable 属性。

> 其实这问题可以这么想，若 exchange 或 queue 未设置 durable 属性，则在其 crash 之后就会无法恢复，那么即使 message 设置了 persistent 属性，仍然存在 message 虽然能恢复但却无处容身的问题。
> 同理，若 message 本身未设置 persistent 属性，则 message 的持久化更无从谈起。

如何确保消息不丢失？

消息持久化的前提是：将交换器/队列的 durable 属性设置为 true ，表示交换器/队列是持久交换器/队列，在服务器崩溃或重启之后不需要重新创建交换器/队列（交换器/队列会自动创建）。

如果消息想要从 RabbitMQ 崩溃中恢复，那么消息必须：

> 在消息发布前，通过把它的 “投递模式” 选项设置为2（持久）来把消息标记成持久化
> 将消息发送到持久交换器
> 消息到达持久队列

RabbitMQ 确保持久性消息能从服务器重启中恢复的方式是，将它们写入磁盘上的一个持久化日志文件。

> 当发布一条持久性消息到持久交换器上时，RabbitMQ 会在消息提交到日志文件后才发送响应（如果消息路由到了非持久队列，它会自动从持久化日志中移除）。
> 一旦消费者从持久队列中消费了一条持久化消息，RabbitMQ 会在持久化日志中把这条消息标记为等待垃圾收集。如果持久化消息在被消费之前 RabbitMQ 重启，那么 RabbitMQ 会自动重建交换器和队列（以及绑定），并重播持久化日志文件中的消息到合适的队列或者交换器上。

什么是死信队列？

DLX，Dead-Letter-Exchange。利用 DLX ，当消息在一个队列中变成死信（dead message）之后，它能被重新 publish 到另一个 Exchange ，这个 Exchange 就是DLX。消息变成死信一向有一下几种情况：

> 消息被拒绝（basic.reject / basic.nack）并且 requeue=false 。
> 消息 TTL 过期（参考：RabbitMQ之TTL（Time-To-Live 过期时间））。
> 队列达到最大长度。

详细的，可以看看 《RabbitMQ 之死信队列》 文章。

“dead letter”queue 的用途？

当消息被 RabbitMQ server 投递到 consumer 后，但 consumer 却通过 Basic.Reject 进行了拒绝时（同时设置 requeue=false），那么该消息会被放入 “dead letter” queue 中。该 queue 可用于排查 message 被 reject 或 undeliver 的原因。

Basic.Reject 的用法是什么？

该信令可用于 consumer 对收到的 message 进行 reject 。

> 若在该信令中设置 requeue=true ，则当 RabbitMQ server 收到该拒绝信令后，会将该 message

重新发送到下一个处于 consume 状态的 consumer 处（理论上仍可能将该消息发送给当前 consumer）。
若设置 requeue=false ，则 RabbitMQ server 在收到拒绝信令后，将直接将该 message 从 queue 中移除。

另外一种移除 queue 中 message 的小技巧是，consumer 回复 Basic.Ack 但不对获取到的 message 做任何处理。而 Basic.Nack是对 Basic.Reject 的扩展，以支持一次拒绝多条 message 的能力。

RabbitMQ 中的 cluster、mirrored queue，以及 warrens 机制分别用于解决什么问题？

1）cluster

cluster 是为了解决当 cluster 中的任意 node 失效后，producer 和 consumer 均可以通过其他 node 继续工作，即提高了可用性；另外可以通过增加 node 数量增加 cluster 的消息吞吐量的目的。
cluster 本身不负责 message 的可靠性问题（该问题由 producer 通过各种机制自行解决）；cluster 无法解决跨数据中心的问题（即脑裂问题）。
另外，在cluster 前使用 HAProxy 可以解决 node 的选择问题，即业务无需知道 cluster 中多个 node 的 ip 地址。可以利用 HAProxy 进行失效 node 的探测，可以作负载均衡。下图为 HAProxy + cluster 的模型：HAProxy + cluster 的模型

2）Mirrored queue

Mirrored queue 是为了解决使用 cluster 时所创建的 queue 的完整信息仅存在于单一 node 上的问题，从另一个角度增加可用性。
若想正确使用该功能，需要保证：1）consumer 需要支持 Consumer Cancellation Notification 机制；2）consumer 必须能够正确处理重复 message 。

3）Warrens

Warrens 是为了解决 cluster 中 message 可能被 blackholed 的问题，即不能接受 producer 不停 republish message 但 RabbitMQ server 无回应的情况。

Warrens 有两种构成方式：

一种模型，是两台独立的 RabbitMQ server + HAProxy ，其中两个 server 的状态分别为 active 和 hot-standby 。该模型的特点为：两台 server 之间无任何数据共享和协议交互，两台 server 可以基于不同的 RabbitMQ 版本。如下图所示：模型一
另一种模型，为两台共享存储的 RabbitMQ server + keepalived ，其中两个 server 的状态分别为 active 和 cold-standby 。该模型的特点为：两台 server 基于共享存储可以做到完全恢复，要求必须基于完全相同的 RabbitMQ 版本。如下图所示：模型二

Warrens 模型存在的问题：

对于第一种模型，虽然理论上讲不会丢失消息，但若在该模型上使用持久化机制，就会出现这样一种情况，即若作为 active 的 server 异常后，持久化在该 server 上的消息将暂时无法被 consume ，因为此时该 queue 将无法在作为 hot-standby 的 server 上被重建，所以，只能等到异常的 active server 恢复后，才能从其上的 queue 中获取相应的 message 进行处理。而对于业务来说，需要具有：a.感知 AMQP 连接断开后重建各种 fabric 的能力；b.感知 active server 恢复的能力；c.切换回 active server 的时机控制，以及切回后，针对 message 先后顺序产生的变化进行处理的能力。
对于第二种模型，因为是基于共享存储的模式，所以导致 active server 异常的条件，可能同样会导致 cold-standby server 异常；另外，在该模型下，要求 active 和 cold-standby

的 server 必须具有相同的 node 名和 UID ，否则将产生访问权限问题；最后，由于该模型是冷备方案，故无法保证 cold-standby server 能在你要求的时限内成功启动。

RabbitMQ 如何实现高可用？

这个问题，和 「RabbitMQ 中的 cluster、mirrored queue，以及 warrens 机制分别用于解决什么问题？」 会比较类似。

RabbitMQ 的高可用，是基于主从做高可用性的。它有三种模式：

单机模式
普通集群模式
镜像集群模式

1）单机模式

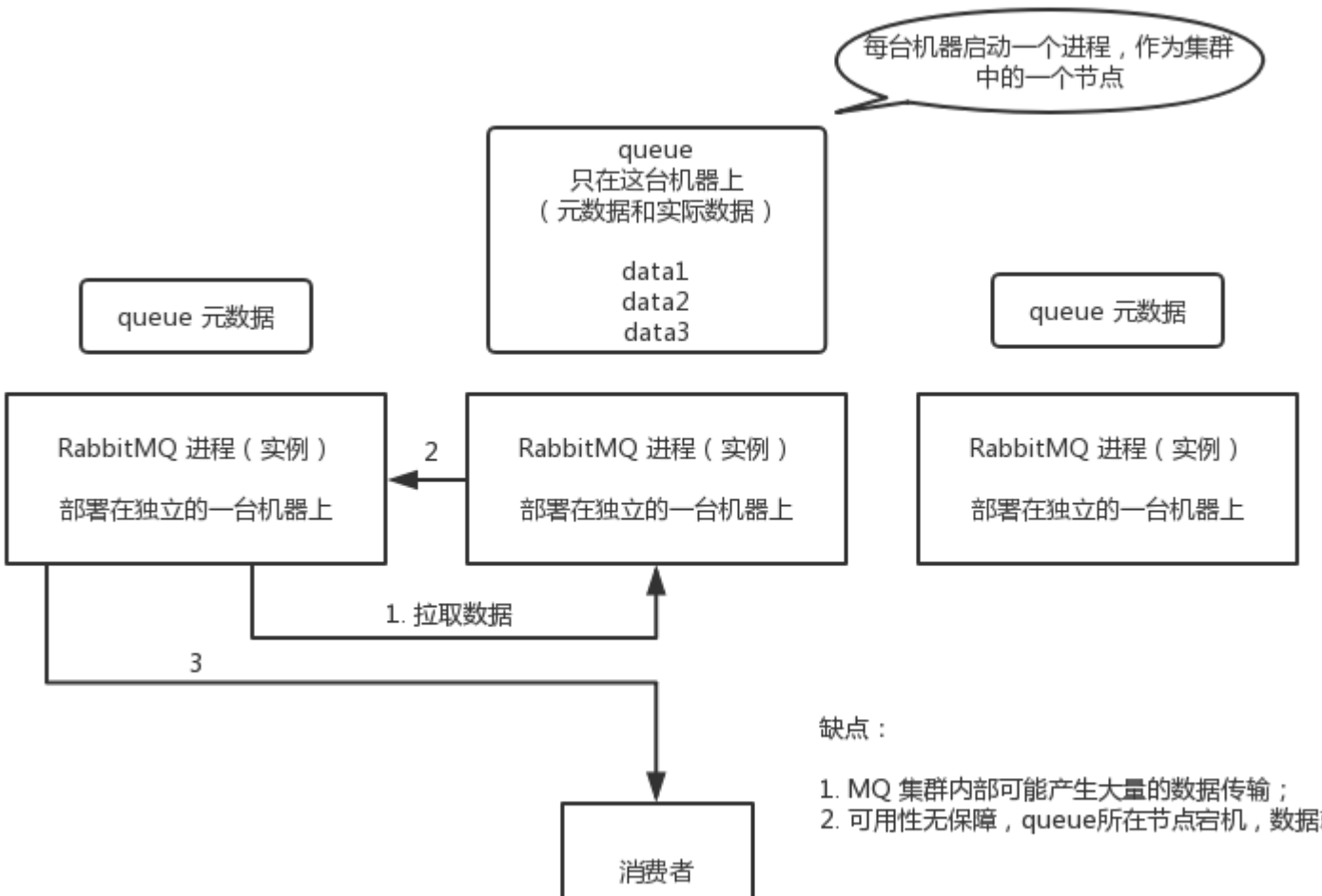
单机模式，就是启动单个 RabbitMQ 节点，一般用于本地开发或者测试环境。实际生产环境下，基本不会使用。

普通集群模式（无高可用性）

这种方式，就是上面问题的 cluster 。

普通集群模式，意思就是在多台机器上启动多个 RabbitMQ 实例，每个机器启动一个。

你创建的 queue，只会放在一个 RabbitMQ 实例上，但是每个实例都同步 queue 的元数据（元数据可以认为是 queue 的一些配置信息，通过元数据，可以找到 queue 所在实例）。你消费的时候，实际上如果连接到了另外一个实例，那么那个实例会从 queue 所在实例上拉取数据过来。



这种方式确实很麻烦，也不怎么好，没做到所谓的分布式，就是个普通集群。因为这导致你要么消费者每次随机连接一个实例然后拉取数据，要么固定连接那个 queue 所在实例消费数据，前者有数据拉取的开销，后者导致单实例性能瓶颈。

而且如果那个放 queue 的实例宕机了，会导致接下来其他实例就无法从那个实例拉取，如果你开启了消息持久化，让 RabbitMQ 落地存储消息的话，消息不一定会丢，得等这个实例恢复了，然后才可以继续从这个 queue 拉取数据。

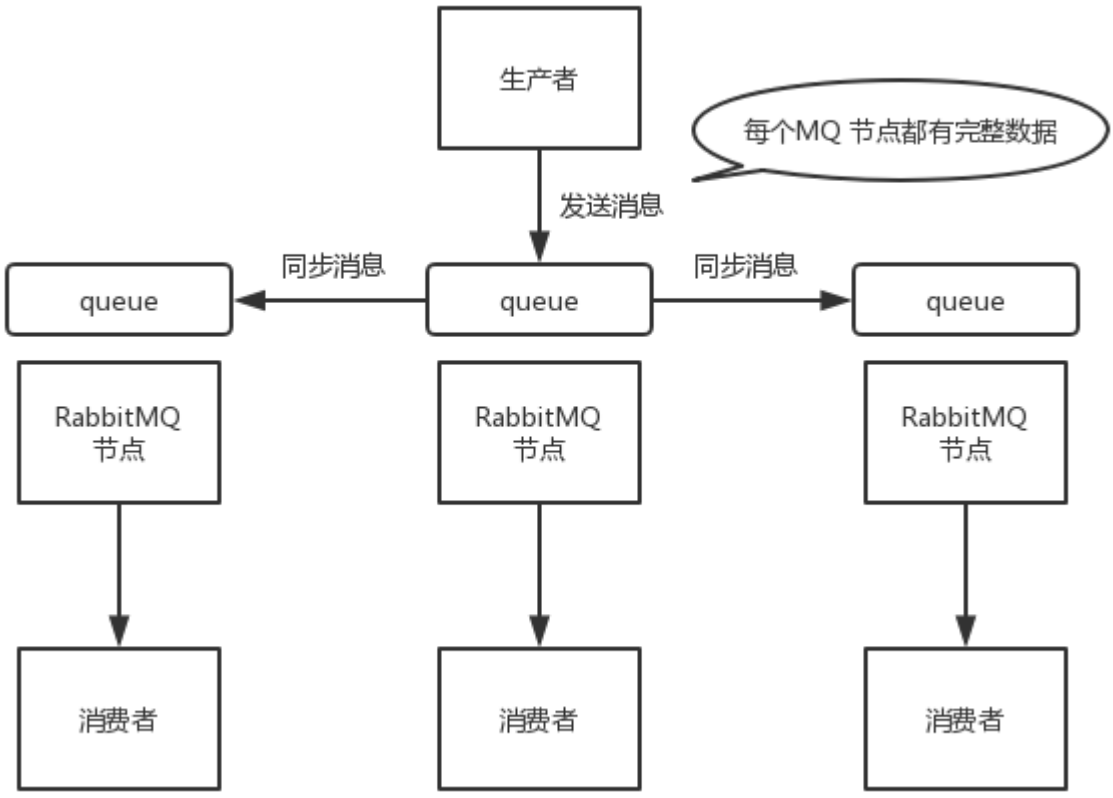
所以这个事儿就比较尴尬了，这就没有什么所谓的高可用性，这方案主要是提高吞吐量的，就是说让集群中多个节点来服务某个 queue 的读写操作。

镜像集群模式（高可用性）

芳芳：请教了下胖友，他们采用这种方式。

这种方式，就是上面问题的 Mirrored queue 。

这种模式，才是所谓的 RabbitMQ 的高可用模式。跟普通集群模式不一样的是，在镜像集群模式下，你创建的 queue，无论元数据还是 queue 里的消息都会存在于多个实例上，就是说，每个 RabbitMQ 节点都有这个 queue 的一个完整镜像，包含 queue 的全部数据的意思。然后每次你写消息到 queue 的时候，都会自动把消息同步到多个实例的 queue 上。



那么如何开启这个镜像集群模式呢？其实很简单，RabbitMQ 有很好的管理控制台，就是在后台新增一个策略，这个策略是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。

这样的话，好处在于，你任何一个机器宕机了，没事儿，其它机器（节点）还包含了这个 queue 的

完整数据，别的 consumer 都可以到其它节点上去消费数据。坏处在于，第一，这个性能开销也太大了吧，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！第二，这么玩儿，不是分布式的，就没有扩展性可言了，如果某个 queue 负载很重，你加机器，新增的机器也包含了这个 queue 的所有数据，并没有办法线性扩展你的 queue。你想，如果这个 queue 的数据量很大，大到这个机器上的容量无法容纳了，此时该怎么办呢？

如何使用 RabbitMQ 实现 RPC

基于 RabbitMQ reply_to 特性，可以很轻易使用 RabbitMQ 实现 RPC 功能。

具体的代码实现，可以看看 《芋道 Spring Boot 消息队列 RabbitMQ 入门》的「15. RPC 远程调用」小节。

使用 RabbitMQ 实现 RPC 有什么好处？

1、将客户端和服务器解耦：客户端只是发布一个请求到 MQ 并消费这个请求的响应。并不关心具体由谁来处理这个请求，MQ 另一端的请求的消费者可以随意替换成任何可以处理请求的服务器，并不影响到客户端。

相当于 RPC 的注册发现功能，交给 RabbitMQ 来实现了。

2、减轻服务器的压力：传统的 RPC 模式中如果客户端和请求过多，服务器的压力会过大。由 MQ 作为中间件的话，过多的请求而是被 MQ 消化掉，服务器可以控制消费请求的频次，并不会影响到服务器。

3、服务器的横向扩展更加容易：如果服务器的处理能力不能满足请求的频次，只需要增加服务器来消费 MQ 的消息即可，MQ会帮我们实现消息消费的负载均衡。

4、可以看出 RabbitMQ 对于 RPC 模式的支持也是比较友好地，amq.rabbitmq.reply-to, reply_to, correlation_id 这些特性都说明了这一点，再加上 spring-rabbit 的实现，可以让我们很简单的使用消息队列模式的 RPC 调用。

例如说：rabbitmq-jsonrpc 的实现。

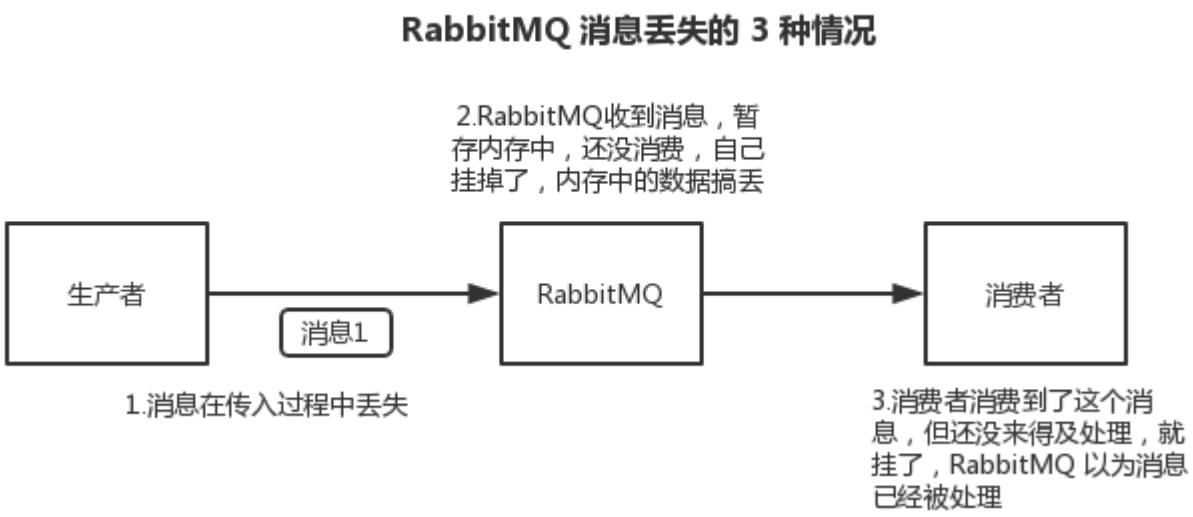
当然，虽然有这些优点，实际场景下，我们并不会这么做。

为什么 heavy RPC 的使用场景下不建议采用 disk node ？

heavy RPC 是指在业务逻辑中高频调用 RabbitMQ 提供的 RPC 机制，导致不断创建、销毁 reply queue ，进而造成 disk node 的性能问题（因为会针对元数据不断写盘）。所以在使用 RPC 机制时需要考虑自身的业务场景，一般来说不建议。

RabbitMQ 是否会弄丢数据？

芳芳：这个问题，基本是我们前面看到的几个问题的总结合并。



生产者弄丢了数据？

生产者将数据发送到 RabbitMQ 的时候，可能数据就在半路给搞丢了，因为网络问题啥的，都有可能。

方案一：事务功能

此时可以选择用 RabbitMQ 提供的【事务功能】，就是生产者发送数据之前开启 RabbitMQ 事务channel.txSelect，然后发送消息，如果消息没有成功被 RabbitMQ 接收到，那么生产者会收到异常报错，此时就可以回滚事务channel.txRollback，然后重试发送消息；如果收到了消息，那么可以提交事务channel.txCommit。代码如下：

```
// 开启事务
channel.txSelect
try {
    // 这里发送消息
} catch (Exception e) {
    channel.txRollback

    // 这里再次重发这条消息
}

// 提交事务
channel.txCommit
```

但是问题是，RabbitMQ 事务机制（同步）一搞，基本上吞吐量会下来，因为太耗性能。

具体的代码实现，可以看看 《芋道 Spring Boot 消息队列 RabbitMQ 入门》的「12. 事务消息」小节。

方案二：confirm 功能。

所以一般来说，如果你要确保说写 RabbitMQ 的消息别丢，可以开启 【confirm 模式】，在生产者那里设置开启 confirm 模式之后，你每次写的消息都会分配一个唯一的 id，然后如果写入了 RabbitMQ 中，RabbitMQ 会给你回传一个 ack 消息，告诉你说这个消息 ok 了。如果 RabbitMQ 没能处理这个消息，会回调你的一个 nack 接口，告诉你这个消息接收失败，你可以重试。而且你可以

结合这个机制自己在内存里维护每个消息 id 的状态，如果超过一定时间还没接收到这个消息的回调，那么你可以重发。

具体的代码实现，可以看看 《芋道 Spring Boot 消息队列 RabbitMQ 入门》的「14. 生产者的发送确认」 小节。

对比总结

事务机制和 confirm 机制最大的不同在于，事务机制是同步的，你提交一个事务之后会阻塞在那儿，但是 confirm 机制是异步的，你发送个消息之后就可以发送下一个消息，然后那个消息 RabbitMQ 接收了之后会异步回调你的一个接口通知你这个消息接收到了。

所以一般在生产者这块避免数据丢失，都是用 confirm 机制的。

> 不过 confirm 功能，也可能存在丢消息的情况。举个例子，如果回调到 nack 接口，此时 JVM 挂掉了，那么此消息就丢失了。（这个是芳芳的猜想，还在找胖友探讨中。关于这块，欢迎星球讨论。）

Broker 弄丢了数据

就是 Broker 自己弄丢了数据，这个你必须开启 Broker 的持久化，就是消息写入之后会持久化到磁盘，哪怕是 Broker 自己挂了，恢复之后会自动读取之前存储的数据，一般数据不会丢。除非极其罕见的是，Broker 还没持久化，自己就挂了，可能导致少量数据丢失，但是这个概率较小。

设置持久化有两个步骤：

> 创建 queue 的时候将其设置为持久化 这样就可以保证 RabbitMQ 持久化 queue 的元数据，但是它是不会持久化 queue 里的数据的。
> 第二个是发送消息的时候将消息的 deliveryMode 设置为 2 就是将消息设置为持久化的，此时 RabbitMQ 就会将消息持久化到磁盘上去。

必须要同时设置这两个持久化才行，Broker 哪怕是挂了，再次重启，也会从磁盘上重启恢复 queue，恢复这个 queue 里的数据。

注意，哪怕是你给 Broker 开启了持久化机制，也有一种可能，就是这个消息写到了 Broker 中，但是还没来得及持久化到磁盘上，结果不巧，此时 Broker 挂了，就会导致内存里的一点点数据丢失。

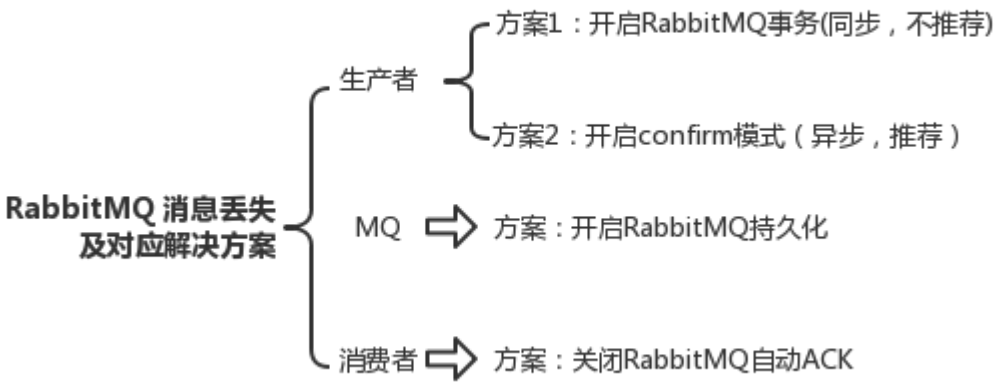
所以，持久化可以跟生产者那边的 confirm 机制配合起来，只有消息被持久化到磁盘之后，才会通知生产者 ack 了，所以哪怕是在持久化到磁盘之前，Broker 挂了，数据丢了，生产者收不到 ack，你也是可以自己重发的。

消费端弄丢了数据？

RabbitMQ 如果丢失了数据，主要是因为你消费的时候，刚消费到，还没处理，结果进程挂了，比如重启了，那么就尴尬了，RabbitMQ 认为你都消费了，这数据就丢了。

这个时候得用 RabbitMQ 提供的 ack 机制，简单来说，就是你必须关闭 RabbitMQ 的自动 ack，可以通过一个 api 来调用就行，然后每次你自己代码里确保处理完的时候，再在程序里 ack 一把。这样的话，如果你还没处理完，不就没有 ack 了？那 RabbitMQ 就认为你还没处理完，这个时候 RabbitMQ 会把这个消费分配给别的 consumer 去处理，消息是不会丢的。

总结

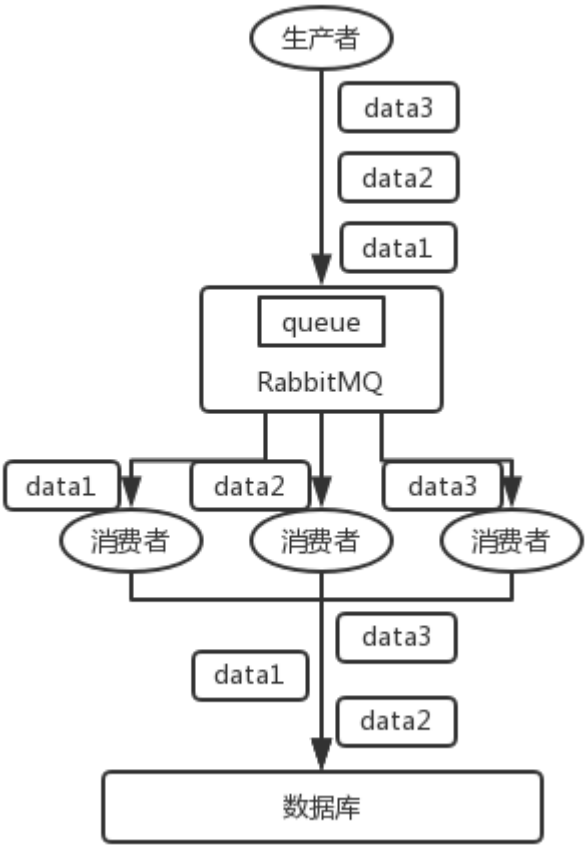


RabbitMQ 如何保证消息的顺序性？

和 Kafka 与 RocketMQ 不同，Kafka 不存在类似类似 Topic 的概念，而是真正的一条一条队列，并且每个队列可以被多个 Consumer 拉取消息。这个，是非常大的一个差异。

来看看 RabbitMQ 顺序错乱的场景：

一个 queue，多个 consumer。比如，生产者向 RabbitMQ 里发送了三条数据，顺序依次是 data1/data2/data3，压入的是 RabbitMQ 的一个内存队列。有三个消费者分别从 MQ 中消费这三条数据中的一条，结果消费者 2 先执行完操作，把 data2 存入数据库，然后是 data1/data3。这不明显乱了。　也就是说，乱序消费的问题。



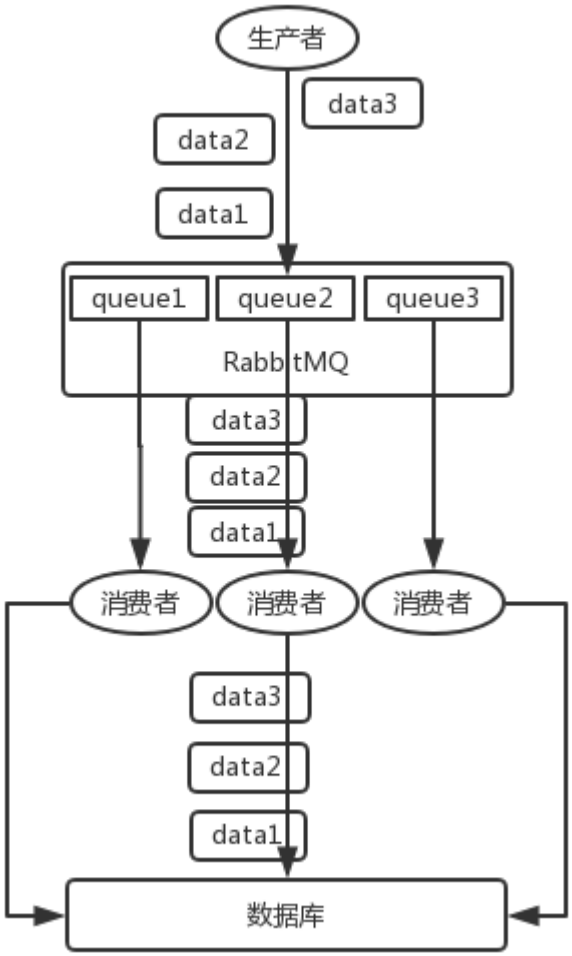
解决方案：

方案一，拆分多个 queue，每个 queue 一个 consumer，就是多一些 queue 而已，确实是麻烦点。

> 这个方式，有点模仿 Kafka 和 RocketMQ 中 Topic 的概念。例如说，原先一个 queue 叫 "xxx"，那么多个 queue，我们可以叫 "xxx-01"、"xxx-02" 等，相同前缀，不同后缀。

方案二，或者就一个 queue 但是对应一个 consumer，然后这个 consumer 内部用内存队列做排队，然后分发给底层不同的 worker 来处理。

> 这种方式，就是讲一个 queue 里的，相同的“key” 交给同一个 worker 来执行。因为 RabbitMQ 是可以单条消息来 ack，所以还是比较方便的。这一点，也是和 RocketMQ 和 Kafka 不同的地方。



实际上，我们会发现上述的两个方案，前提都是一个 queue 只能启动一个 consumer 对应。

具体的代码实现，可以看看 《芋道 Spring Boot 消息队列 RabbitMQ 入门》的「11. 顺序消息」小节

666. 彩蛋

写的很糟糕，一度想删除。后来想想，先就酱紫，可能这就是此时自己对 RabbitMQ 掌握的情况。后面等自己业务场景真的开始使用 RabbitMQ 之后，在好好倒腾倒腾。Sad But Tree 。

参考与推荐如下文章：

> 《RabbitMQ 面试专题》
> 《如何保证消息队列的高可用？》
> 《RabbitMQ 面试要点》
> 《如何保证消息的可靠性传输？（如何处理消息丢失的问题）》

文章目录

回到首页