干货：这也许是最全面透彻的一篇RabbitMQ指南！

2017-08-24 高广超 DBAplus社群

作者介绍

高广超，目前就职于美团网，负责美团点评外卖核心业务的后台研发工作。多年一线互联网研发与架构设计经验，擅长高可用、高性能互联网架构的设计与落地。个人博客地址：http://www.jianshu.com/u/2766e4cfc391。

本文大纲：

历史-从开始到现在

RabbitMQ是一个Erlang开发的AMQP（Advanced Message Queuing Protocol ）的开源实现。AMQP 的出现其实也是应了广大人民群众的需求，虽然在同步消息通讯的世界里有很多公开标准（如 Cobar）的 IIOP ，或者是 SOAP 等），但是在异步消息处理中却不是这样，只有大企业有一些商业实现（如微软的 MSMQ ，IBM 的 WebSphere MQ 等），因此，在 2006 年的 6 月，Cisco 、Red Hat、iMatix 等联合制定了 AMQP 的公开标准。

RabbitMQ由RabbitMQ Technologies Ltd开发并且提供商业支持的。该公司在2010年4月被SpringSource （VMware的一个部门）收购。在2013年5月被并入Pivotal。其实VMware，Pivotal和EMC本质上是一家的。不同的是，VMware是独立上市子公司，而Pivotal是整合了EMC的某些资源，现在并没有上市。

RabbitMQ官网：http://www.rabbitmq.com

一、应用场景

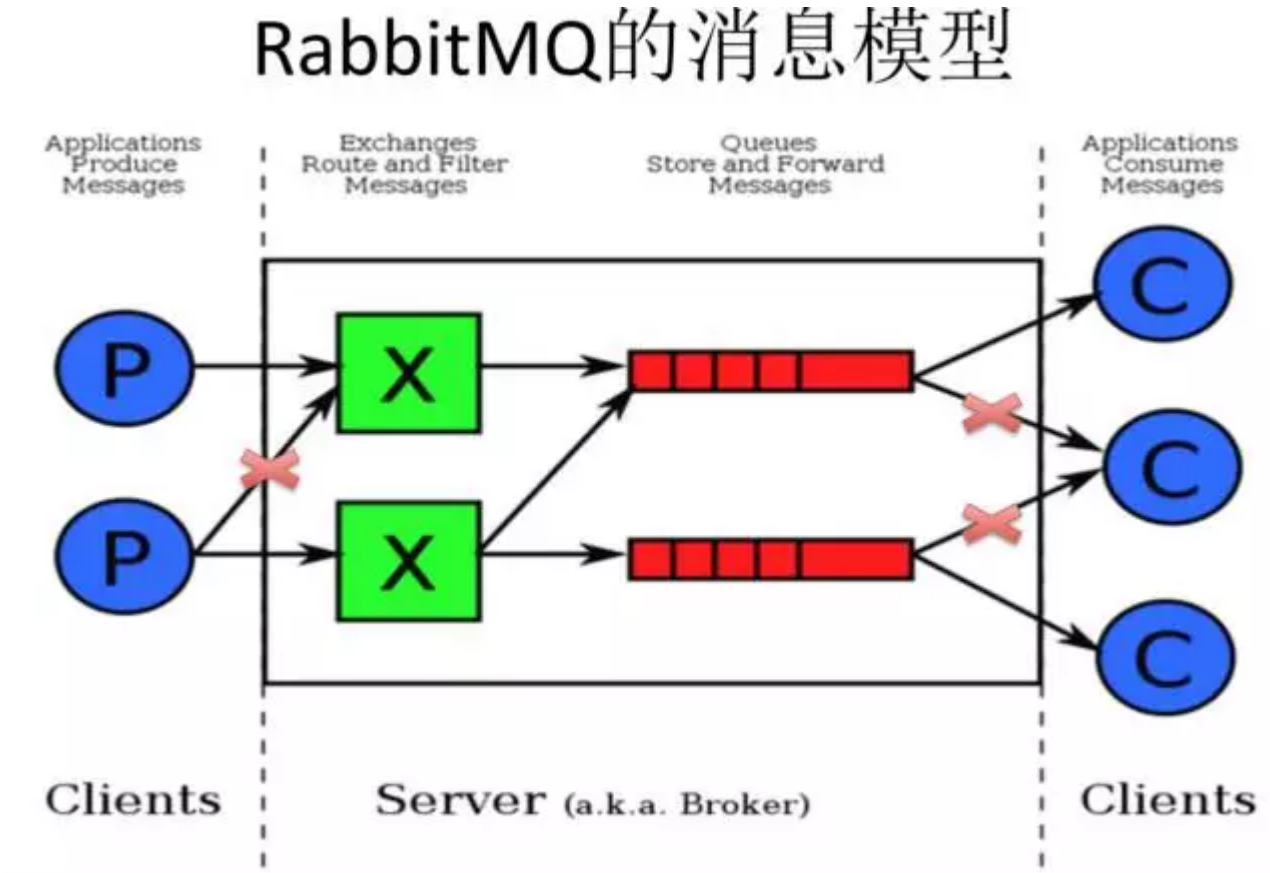
言归正传。RabbitMQ，或者说AMQP解决了什么问题，或者说它的应用场景是什么？

对于一个大型的软件系统来说，它会有很多的组件或者说模块，又或者说子系统。那这些模块又如何通信？这和传统的IPC有很大的区别。传统的IPC很多都是在单一系统上的，模块耦合性很大，不适合扩展（Scalability）。如果使用Socket，那么不同的模块的确可以部署到不同的机器上，但是还是有很多问题需要解决。比如：

- 信息的发送者和接收者如何维持这个连接，如果一方的连接中断，这期间的数据是以什么方式丢失？
- 如何降低发送者和接收者的耦合度？
- 如何让Priority高的接收者先接到数据？
- 如何做到Load Balance？有效均衡接收者的负载？
- 如何有效的将数据发送到相关的接收者？也就是说将接收者subscribe 不同的数据，如何做有效的filter。
- 如何做到可扩展，甚至将这个通信模块发到cluster上？
- 如何保证接收者接收到了完整，正确的数据？

AMQP协议解决了以上的问题，而RabbitMQ实现了AMQP。

二、系统架构



RabbitMQ Server

也叫Broker Server，它不是运送食物的卡车，而是一种传输服务。原话是RabbitMQ isn't a food truck, it's a delivery service. 它的角色就是维护一条从Producer到Consumer的路线，保证数据能够按照指定的方式进行传输。虽然这个保证也不是100%的保证，但是对于普通的应用来说这已经足够了。当然对于商业系统来说，可以再做一层数据一致性的guard，就可以彻底保证系统的一致性了。

Client P

也叫Producer，数据的发送方。Create messages and publish (send) them to a Broker Server (RabbitMQ)。一个Message有两个部分：payload（有效载荷）和label（标签）。payload顾名思义就是传输的数据。label是exchange的名字或者说是一个tag，它描述了payload，而且RabbitMQ也是通过这个label来决定把这个Message发给哪个Consumer。AMQP仅仅描述了label，而RabbitMQ决定了如何使用这个label的规则。

Client C

也叫Consumer，数据的接收方。Consumers attach to a Broker Server (RabbitMQ) and subscribe to a queue。把queue比作是一个有名字的邮箱。当有Message到达某个邮箱后，RabbitMQ把它发送给它的某个订阅者即Consumer。当然可能会把同一个Message发送给很多的Consumer。在这个Message中，只有payload，label已经被删掉了。对于Consumer来说，

它是不知道谁发送的这个信息的,就是协议本身不支持。当然了,如果Producer发送的payload包含了Producer的信息就另当别论了。

对于一个数据从Producer到Consumer的正确传递，还有三个概念需要明确： exchanges, queues and bindings。

- Exchanges are where producers publish their messages.
- Queues are where the messages end up and are received by consumers.
- Bindings are how the messages get routed from the exchange to particular queues.

还有几个概念是上述图中没有标明的，那就是Connection（连接）和Channel（通道，频道）。

Connection

就是一个TCP的连接。Producer和Consumer都是通过TCP连接到RabbitMQ Server的。以后我们可以看到，程序的起始处就是建立这个TCP连接。

Channel

虚拟连接。它建立在上述的TCP连接中。数据流动都是在Channel中进行的。也就是说，一般情况是程序起始建立TCP连接，第二步就是建立这个Channel。

那么，为什么使用Channel，而不是直接使用TCP连接?

对于OS来说，建立和关闭TCP连接是有代价的，频繁的建立关闭TCP连接对于系统的性能有很大的影响，而且TCP的连接数也有限制，这也限制了系统处理高并发的能力。但是，在TCP连接中建立Channel是没有上述代价的。对于Producer或者Consumer来说，可以并发的使用多个Channel进行Publish或者Receive。有实验表明，1s的数据可以Publish10K的数据包。当然对于不同的硬件环境，不同的数据包大小这个数据肯定不一样，但是我只想说明，对于普通的Consumer或者Producer来说，这已经足够了。如果不够用，你考虑的应该是如何细化SPLIT你的设计。

相关定义：

- Broker： 简单来说就是消息队列服务器实体
- Exchange： 消息交换机，它指定消息按什么规则，路由到哪个队列
- Queue： 消息队列载体，每个消息都会被投入到一个或多个队列
- Binding： 绑定，它的作用就是把exchange和queue按照路由规则绑定起来
- Routing Key： 路由关键字，exchange根据这个关键字进行消息投递
- VHost： 虚拟主机，一个broker里可以开设多个vhost，用作不同用户的权限分离。
- Producer： 消息生产者，就是投递消息的程序
- Consumer： 消息消费者，就是接受消息的程序

- Channel： 消息通道，在客户端的每个连接里，可建立多个channel，每个channel代表一个会话任务

由Exchange、Queue、RoutingKey三个才能决定一个从Exchange到Queue的唯一的线路。

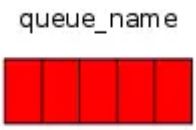
三、基本概念

Connection Factory、Connection、Channel都是RabbitMQ对外提供的API中最基本的对象。Connection是RabbitMQ的socket链接，它封装了socket协议相关部分逻辑。Connection Factory则是Connection的制造工厂。

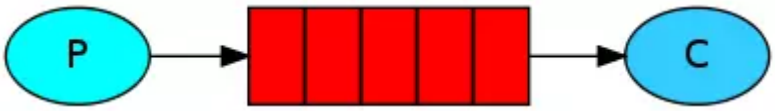
Channel是我们与RabbitMQ打交道的最重要的一个接口，我们大部分的业务操作是在Channel这个接口中完成的，包括定义Queue、定义Exchange、绑定Queue与Exchange、发布消息等。

Queue

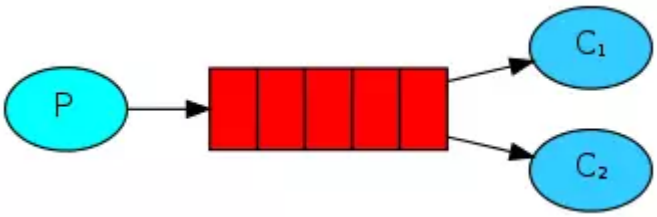
Queue（队列）是RabbitMQ的内部对象，用于存储消息，如下图表示。



RabbitMQ中的消息都只能存储在Queue中，生产者（下图中的P）生产消息并最终投递到Queue中，消费者（下图中的C）可以从Queue中获取消息并消费。



多个消费者可以订阅同一个Queue，这时Queue中的消息会被平均分摊给多个消费者进行处理，而不是每个消费者都收到所有的消息并处理。



Message acknowledgment

在实际应用中，可能会发生消费者收到Queue中的消息，但没有处理完成就宕机（或出现其他意外）的情况，这种情况下就可能会导致消息丢失。为了避免这种情况发生，我们可以要求消费者在消费完消息后发送一个回执给RabbitMQ，RabbitMQ收到消息回执（Message acknowledgment）后才将该消息从Queue中移除。

如果RabbitMQ没有收到回执并检测到消费者的RabbitMQ连接断开，则RabbitMQ会将该消息发送给其他消费者（如果存在多个消费者）进行处理。这里不存在timeout，一个消费者处理消息时间再长也不会导致该消息被发送给其他消费者，除非它的RabbitMQ连接断开。

这里会产生另外一个问题，如果我们的开发人员在处理完业务逻辑后，忘记发送回执给RabbitMQ，这将会导致严重的bug——Queue中堆积的消息会越来越多。消费者重启后会重复消费这些消息并重复执行业务逻辑。

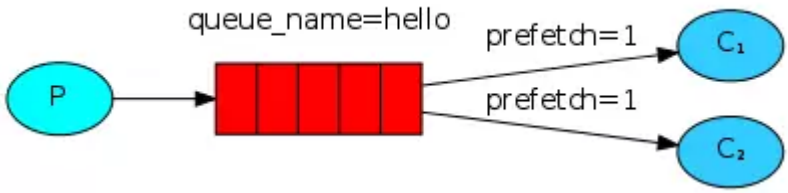
另外publish message 是没有ACK的。

Message durability

如果我们希望即使在RabbitMQ服务重启的情况下，也不会丢失消息，我们可以将Queue与Message都设置为可持久化的（durable），这样可以保证绝大部分情况下我们的RabbitMQ消息不会丢失。但依然解决不了小概率丢失事件的发生（比如RabbitMQ服务器已经接收到生产者的消息，但还没来得及持久化该消息时RabbitMQ服务器就断电了），如果我们需要对这种小概率事件也要管理起来，那么我们要用到事务。由于这里仅为RabbitMQ的简单介绍，所以这里将不讲解RabbitMQ相关的事务。

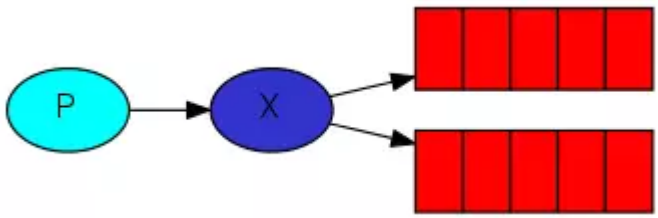
Prefetch count

前面我们讲到如果有多个消费者同时订阅同一个Queue中的消息，Queue中的消息会被平摊给多个消费者。这时如果每个消息的处理时间不同，就有可能会导致某些消费者一直在忙，而另外一些消费者很快就处理完手头工作并一直空闲的情况。我们可以通过设置Prefetch count来限制Queue每次发送给每个消费者的消息数，比如我们设置prefetchCount=1，则Queue每次给每个消费者发送一条消息；消费者处理完这条消息后Queue会再给该消费者发送一条消息。



Exchange

在上一节我们看到生产者将消息投递到Queue中，实际上这在RabbitMQ中这种事情永远都不会发生。实际的情况是，生产者将消息发送到Exchange（交换器，下图中的X），由Exchange将消息路由到一个或多个Queue中（或者丢弃）。



Exchange是按照什么逻辑将消息路由到Queue的？这个将在Binding一节中介绍。

RabbitMQ中的Exchange有四种类型，不同的类型有着不同的路由策略，这将在Exchange Types一节介绍。

Routing Key

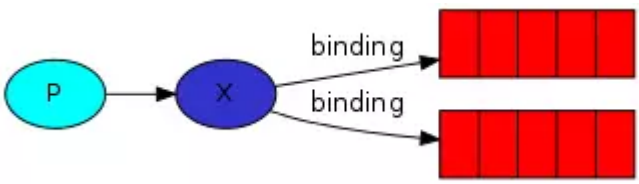
生产者在将消息发送给Exchange的时候，一般会指定一个Routing Key，来指定这个消息的路由规则，而这个Routing Key需要与Exchange Type及Binding key联合使用才能最终生效。

在Exchange Type与Binding key固定的情况下（在正常使用时一般这些内容都是固定配置好的），我们的生产者就可以在发送消息给Exchange时，通过指定Routing Key来决定消息流向哪里。

RabbitMQ为Routing Key设定的长度限制为255 bytes。

Binding

RabbitMQ中通过Binding将Exchange与Queue关联起来，这样RabbitMQ就知道如何正确地将消息路由到指定的Queue了。



Binding key

在绑定（Binding）Exchange与Queue的同时，一般会指定一个Binding key。消费者将消息发送给Exchange时，一般会指定一个Routing Key。当Binding key与Routing Key相匹配时，消息将会被路由到对应的Queue中。这个将在Exchange Types章节会列举实际的例子加以说明。

在绑定多个Queue到同一个Exchange的时候，这些Binding允许使用相同的Binding key。

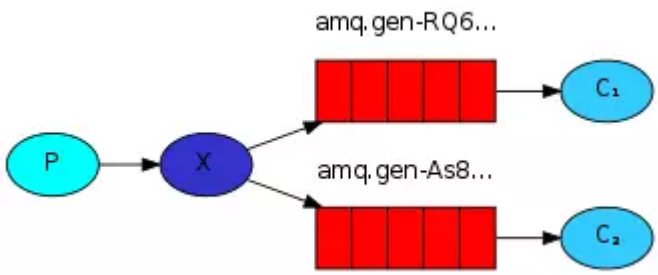
Binding key并不是在所有情况下都生效，它依赖于Exchange Type，比如fanout类型的Exchange就会无视Binding key，而是将消息路由到所有绑定到该Exchange的Queue。

Exchange Types

RabbitMQ常用的Exchange Type有fanout、direct、topic、headers这四种（AMQP规范里还提到两种Exchange Type，分别为system与自定义，这里不予以描述），下面分别进行介绍。

fanout

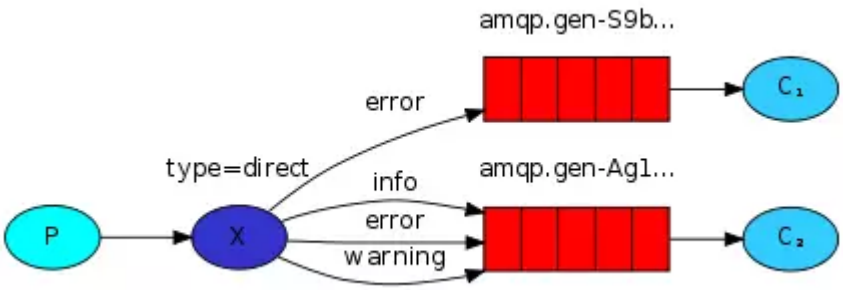
fanout类型的Exchange路由规则非常简单，它会把所有发送到该Exchange的消息路由到所有与它绑定的Queue中。



上图中，生产者（P）发送到Exchange（X）的所有消息都会路由到图中的两个Queue，并最终被两个消费者（C1与C2）消费。

direct

direct类型的Exchange路由规则也很简单，它会把消息路由到那些Binding key与Routing key完全匹配的Queue中。



以上图的配置为例，我们以routingKey="error"发送消息到Exchange，则消息会路由到Queue1（amqp.gen-S9b...，这是由RabbitMQ自动生成的Queue名称）和

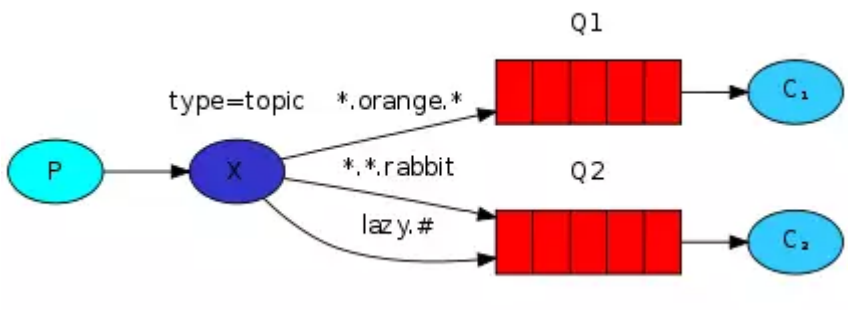
Queue2（amqp.gen-Agl...）；如果我们以Routing Key="info"或routingKey="warning"来发送消息，则消息只会路由到Queue2。如果我们以其他Routing Key发送消息，则消息不会路由到这两个Queue中。

topic

前面讲到direct类型的Exchange路由规则是完全匹配Binding Key与Routing Key，但这种严格的匹配方式在很多情况下不能满足实际业务需求。topic类型的Exchange在匹配规则上进行了扩展，它与direct类型的Exchage相似，也是将消息路由到Binding Key与Routing Key相匹配的Queue中，但这里的匹配规则有些不同，它约定：

Routing Key为一个句点号“.”分隔的字符串（我们将被句点号"."分隔开的每一段独立的字符串称为一个单词），如"stock.usd.nyse"、"nyse.vmw"、"quick.orange.rabbit"。Binding Key与Routing Key一样也是句点号“.”分隔的字符串。

Binding Key中可以存在两种特殊字符"*"与"#"，用于做模糊匹配，其中"*"用于匹配一个单词，"#"用于匹配多个单词（可以是零个）。



以上图中的配置为例，routingKey=”quick.orange.rabbit”的消息会同时路由到Q1与Q2，routingKey=”lazy.orange.fox”的消息会路由到Q1，routingKey=”lazy.brown.fox”的消息会路由到Q2，routingKey=”lazy.pink.rabbit”的消息会路由到Q2（只会投递给Q2一次，虽然这个routingKey与Q2的两个bindingKey都匹配）；routingKey=”quick.brown.fox”、routingKey=”orange”、routingKey=”quick.orange.male.rabbit”的消息将会被丢弃，因为它们没有匹配任何bindingKey。

headers

headers类型的Exchange不依赖于Routing Key与Binding Key的匹配规则来路由消息，而是根据发送的消息内容中的headers属性进行匹配。

在绑定Queue与Exchange时指定一组键值对；当消息发送到Exchange时，RabbitMQ会取到该消息的headers（也是一个键值对的形式），对比其中的键值对是否完全匹配Queue与

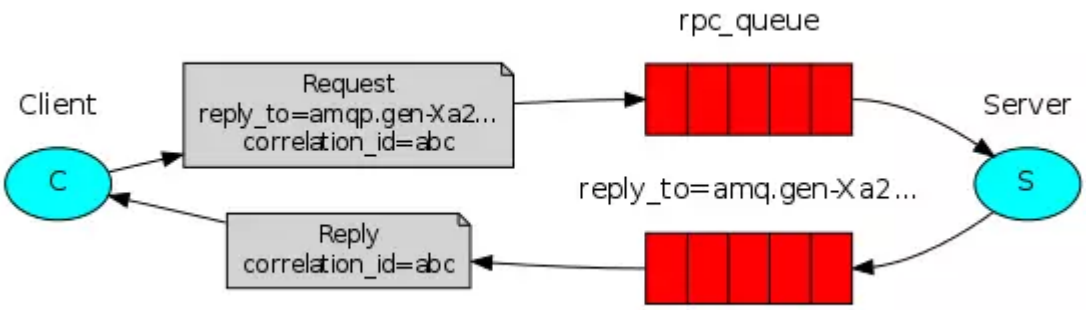
Exchange绑定时指定的键值对。如果完全匹配则消息会路由到该Queue，否则不会路由到该Queue。

该类型的Exchange没有用到过（不过也应该很有用武之地），所以不做介绍。

RPC

MQ本身是基于异步的消息处理，前面的示例中所有的生产者（P）将消息发送到RabbitMQ后不会知道消费者（C）处理成功或者失败（甚至连有没有消费者来处理这条消息都不知道）。

但实际的应用场景中，我们很可能需要一些同步处理，需要同步等待服务端将我的消息处理完成后再进行下一步处理。这相当于RPC（Remote Procedure Call，远程过程调用）。在RabbitMQ中也支持RPC。



RabbitMQ**中实现**RPC**的机制是**:

客户端发送请求（消息）时，在消息的属性（Message Properties，在AMQP协议中定义了14种properties，这些属性会随着消息一起发送）中设置两个值replyTo（一个Queue名称，用于告诉服务器处理完成后将通知我的消息发送到这个Queue中）和correlationId（此次请求的标识号，服务器处理完成后需要将此属性返还，客户端将根据这个id了解哪条请求被成功执行了或执行失败）。服务器端收到消息处理完后，将生成一条应答消息到replyTo指定的Queue，同时带上correlationId属性。客户端之前已订阅replyTo指定的Queue，从中收到服务器的应答消息后，根据其中的correlationId属性分析哪条请求被执行了，根据执行结果进行后续业务处理。

四、细节阐明

使用ACK确认Message的正确传递

默认情况下，如果Message 已经被某个Consumer正确的接收到了，那么该Message就会被从Queue中移除。当然也可以让同一个Message发送到很多的Consumer。

如果一个Queue没被任何的Consumer Subscribe（订阅），当有数据到达时，这个数据会被cache，不会被丢弃。当有Consumer时，这个数据会被立即发送到这个Consumer。这个数据被Consumer正确收到时，这个数据就被从Queue中删除。

那么什么是正确收到呢？通过ACK。每个Message都要被acknowledged（确认，ACK）。我们可以显示的在程序中去ACK，也可以自动的ACK。如果有数据没有被ACK，那么RabbitMQ Server会把这个信息发送到下一个Consumer。

如果这个APP有bug，忘记了ACK，那么RabbitMQ Server不会再发送数据给它，因为Server认为这个Consumer处理能力有限。而且ACK的机制可以起到限流的作用（Benefitto throttling）：在Consumer处理完成数据后发送ACK，甚至在额外的延时后发送ACK，将有效的balance Consumer的load。

当然对于实际的例子，比如我们可能会对某些数据进行merge，比如merge 4s内的数据，然后sleep 4s后再获取数据。特别是在监听系统的state，我们不希望所有的state实时的传递上去，而是希望有一定的延时。这样可以减少某些IO，而且终端用户也不会感觉到。

Reject a message

有两种方式，第一种的Reject可以让RabbitMQ Server将该Message 发送到下一个Consumer。第二种是从Queue中立即删除该Message。

Creating a queue

Consumer和Procuder都可以通过 queue.declare 创建queue。对于某个Channel来说，Consumer不能declare一个queue，却订阅其他的queue。当然也可以创建私有的queue。这样只有APP本身才可以使用这个queue。queue也可以自动删除，被标为auto-delete的queue在最后一个Consumer unsubscribe后就会被自动删除。那么如果是创建一个已经存在的queue呢？那么不会有任何的影响。需要注意的是没有任何的影响，也就是说第二次创建如果参数和第一次不一样，那么该操作虽然成功，但是queue的属性并不会被修改。

那么谁应该负责创建这个queue呢？是Consumer，还是Producer?

如果queue不存在，当然Consumer不会得到任何的Message。那么Producer Publish的Message会被丢弃。所以，还是为了数据不丢失，Consumer和Producer都try to create the queue！反正不管怎么样，这个接口都不会出问题。

queue对load balance的处理是完美的。对于多个Consumer来说，RabbitMQ 使用循环的方式（round-robin）的方式均衡的发送给不同的Consumer。

Exchanges

从架构图可以看出，Procuder Publish的Message进入了Exchange。接着通过"routing keys”， RabbitMQ会找到应该把这个Message放到哪个queue里。queue也是通过这个routing keys来做的绑定。

有三种类型的Exchanges：direct, fanout,topic。 每个实现了不同的路由算法（routing algorithm）。

- Direct exchange：如果 routing key 匹配，那么Message就会被传递到相应的queue中。其实在queue创建时，它会自动的以queue的名字作为routing key来绑定那个exchange。
- Fanout exchange： 会向响应的queue广播。
- Topic exchange：对key进行模式匹配，比如ab可以传递到所有ab的queue。

Virtual hosts

每个virtual host本质上都是一个RabbitMQ Server，拥有它自己的queue，exchagne，和bings rule等等。这保证了你可以在多个不同的Application中使用RabbitMQ。

-END-

盛世美颜的人千篇一律

身怀绝技的人万里挑一