<https://www.sojson.com/blog/48.html>

前言：

    MQ 是什么？队列是什么，MQ 我们可以理解为消息队列，队列我们可以理解为管道。以管道的方式做消息传递。

场景：

    1.其实我们在双11的时候，当我们凌晨大量的秒杀和抢购商品，然后去结算的时候，就会发现，界面会提醒我们，让我们稍等，以及一些友好的图片文字提醒。而不是像前几年的时代，动不动就页面卡死，报错等来呈现给用户。

    在这业务场景中，我们就可以采用队列的机制来处理，因为同时结算就只能达到这么多。

    2.在我们平时的超市中购物也是一样，当我们在结算的时候，并不会一窝蜂一样涌入收银台，而是排队结算。这也是队列机制。

对，就是排队。一个接着一个的处理，不能插队。

## RabbitMQ简介

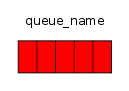
AMQP，即Advanced Message Queuing Protocol，高级消息队列协议，是应用层协议的一个开放标准，为面向消息的中间件设计。消息中间件主要用于组件之间的解耦，消息的发送者无需知道消息使用者的存在，反之亦然。 AMQP的主要特征是面向消息、队列、路由（包括点对点和发布/订阅）、可靠性、安全。 RabbitMQ是一个开源的AMQP实现，服务器端用Erlang语言编写，支持多种客户端，如：Python、Ruby、.NET、Java、JMS、C、PHP、ActionScript、XMPP、STOMP等，支持AJAX。用于在分布式系统中存储转发消息，在易用性、扩展性、高可用性等方面表现不俗。 下面将重点介绍RabbitMQ中的一些基础概念，了解了这些概念，是使用好RabbitMQ的基础。

### ConnectionFactory、Connection、Channel

ConnectionFactory、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。ConnectionFactory为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。

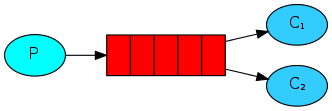
### Queue

Queue（队列）是RabbitMQ的内部对象，用于存储消息，用下图表示。

[](http://ostest.qiniudn.com/wordpress/wp-content/uploads/2014/02/queue.png)

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。

[qq](http://ostest.qiniudn.com/wordpress/wp-content/uploads/2014/02/qq.png)

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。 [](http://ostest.qiniudn.com/wordpress/wp-content/uploads/2014/02/2014-2-21-9-46-43.png)

### 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概念，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…

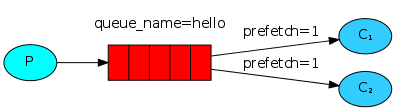
另外pub message是没有ack的。

### Message durability

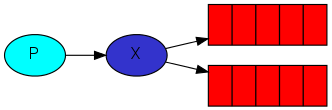
如果我们希望即使在RabbitMQ服务重启的情况下，也不会丢失消息，我们可以将Queue与Message都设置为可持久化的（durable），这样可以保证绝大部分情况下我们的RabbitMQ消息不会丢失。但依然解决不了小概率丢失事件的发生（比如RabbitMQ服务器已经接收到生产者的消息，但还没来得及持久化该消息时RabbitMQ服务器就断电了），如果我们需要对这种小概率事件也要管理起来，那么我们要用到事务。由于这里仅为RabbitMQ的简单介绍，所以这里将不讲解RabbitMQ相关的事务。

### 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prefetchCount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。

[](http://ostest.qiniudn.com/wordpress/wp-content/uploads/2014/02/2014-2-21-9-49-08.png)

### Exchange

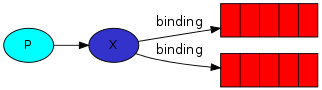
在上一节我们看到生产者将消息投递到Queue中，实际上这在RabbitMQ中这种事情永远都不会发生。实际的情况是，生产者将消息发送到Exchange（交换器，下图中的X），由Exchange将消息路由到一个或多个Queue中（或者丢弃）。 [](http://ostest.qiniudn.com/wordpress/wp-content/uploads/2014/02/2014-2-21-9-51-03.png)

Exchange是按照什么逻辑将消息路由到Queue的？这个将在Binding一节介绍。 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。 [](http://ostest.qiniudn.com/wordpress/wp-content/uploads/2014/02/2014-2-21-9-52-46.png)

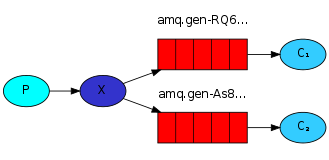
### Binding key

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

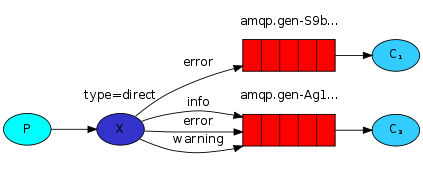
RabbitMQ常用的Exchange Type有fanout、direct、topic、headers这四种（AMQP规范里还提到两种Exchange Type，分别为system与自定义，这里不予以描述），下面分别进行介绍。

### 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。 [](http://ostest.qiniudn.com/wordpress/wp-content/uploads/2014/02/2014-2-21-9-54-26.png)

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

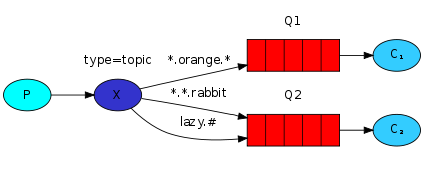
direct类型的Exchange路由规则也很简单，它会把消息路由到那些binding key与routing key完全匹配的Queue中。 [](http://ostest.qiniudn.com/wordpress/wp-content/uploads/2014/02/2014-2-21-9-55-20.png)

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routingKey=”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routingKey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）

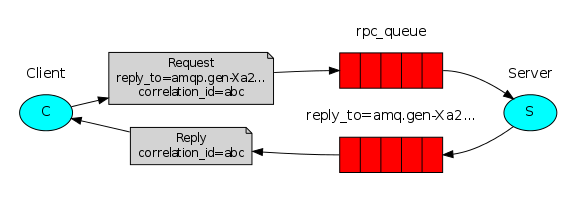
[](http://ostest.qiniudn.com/wordpress/wp-content/uploads/2014/02/2014-2-21-9-57-37.png)

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与Q2，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。 [](http://ostest.qiniudn.com/wordpress/wp-content/uploads/2014/02/2014-2-21-9-59-04.png)

[RabbitMQ](http://www.sojson.com/tag_rabbitmq.html" \o "RabbitMQ" \t "_blank)  中实现RPC 的机制是：

* 客户端发送请求（消息）时，在消息的属性（MessageProperties ，在AMQP 协议中定义了14中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）
* 服务器端收到消息并处理
* 服务器端处理完消息后，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

### 总结

本文介绍了RabbitMQ 中个人认为最重要的概念，充分利用RabbitMQ 提供的这些功能就可以处理我们绝大部分的异步业务了。

## RabbitMQ 选型和对比

#### 1.从社区活跃度

按照目前网络上的资料，RabbitMQ 、activeM 、ZeroMQ 三者中，综合来看，RabbitMQ 是首选。

#### 2.持久化消息比较

ZeroMq 不支持，ActiveMq 和RabbitMq 都支持。持久化消息主要是指我们机器在不可抗力因素等情况下挂掉了，消息不会丢失的机制。

#### 3.综合技术实现

可靠性、灵活的路由、集群、事务、高可用的队列、消息排序、问题追踪、可视化管理工具、插件系统等等。

RabbitMq / Kafka 最好，ActiveMq 次之，ZeroMq 最差。当然ZeroMq 也可以做到，不过自己必须手动写代码实现，代码量不小。尤其是可靠性中的：持久性、投递确认、发布者证实和高可用性。

#### 4.高并发

毋庸置疑，RabbitMQ 最高，原因是它的实现语言是天生具备高并发高可用的erlang 语言。

#### 5.比较关注的比较， RabbitMQ 和 Kafka

RabbitMq 比Kafka 成熟，在可用性上，稳定性上，可靠性上，  [RabbitMq](http://www.sojson.com/tag_rabbitmq.html" \o "RabbitMq" \t "_blank)  胜于  [Kafka](http://www.sojson.com/tag_kafka.html)  （理论上）。

另外，Kafka 的定位主要在日志等方面， 因为Kafka 设计的初衷就是处理日志的，可以看做是一个日志（消息）系统一个重要组件，针对性很强，所以 如果业务方面还是建议选择 RabbitMq 。

还有就是，Kafka 的性能（吞吐量、TPS ）比RabbitMq 要高出来很多。

## 选型最后总结：

如果我们系统中已经有选择  [Kafka](http://www.sojson.com/tag_kafka.html)  ，或者   [RabbitMq](http://www.sojson.com/tag_rabbitmq.html" \o "RabbitMq" \t "_blank)  ，并且完全可以满足现在的业务，建议就不用重复去增加和造轮子。

可以在  [Kafka](http://www.sojson.com/tag_kafka.html)  和   [RabbitMq](http://www.sojson.com/tag_rabbitmq.html" \o "RabbitMq" \t "_blank)  中选择一个适合自己团队和业务的，这个才是最重要的。但是毋庸置疑现阶段，综合考虑没有第三选择。