RabbitMQ的应用场景以及基本原理介绍

# 背景

**MQ** 是什么？**队列**是什么，MQ(Message Queue) 我们可以理解为**消息队列**，队列我们可以理解为**管道**(Channel)。以管道的方式做**消息传递**。

RabbitMQ是一个由**erlang**开发的AMQP，(即Advanced Message Queuing Protocol，**高级消息队列协议**)的开源实现。

AMQP简介：

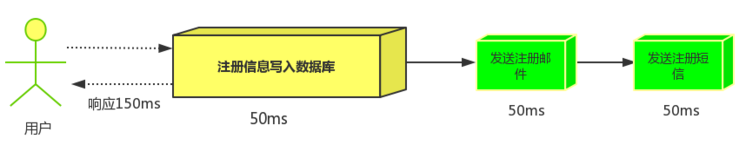
AMQP，即Advanced Message Queuing Protocol，高级消息队列协议，是应用层协议的一个开放标准，为**面向消息的中间件设计**。**消息中间件**主要用于组件之间的解耦，消息的发送者无需知道消息使用者的存在，反之亦然。 **AMQP的主要特征是面向消息、队列、路由（包括点对点和发布/订阅）、可靠性、安全。** RabbitMQ是一个开源的AMQP实现，服务器端用Erlang语言编写，支持多种客户端，如：Python、Ruby、.NET、Java、JMS、C、PHP、ActionScript、XMPP、STOMP等，支持AJAX。用于在分布式系统中存储转发消息，在易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

# 应用场景分析

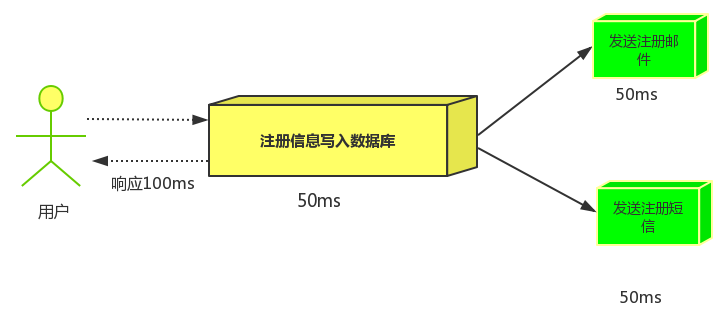
## 异步处理

场景说明：用户注册后，需要发注册邮件和注册短信,传统的做法有两种1.串行的方式;2.并行的方式

1. **串行方式**:将注册信息写入数据库后,发送注册邮件,再发送注册短信,以上三个任务全部完成后才返回给客户端。 这有一个问题是,邮件,短信并不是必须的,它只是一个通知,而这种做法让客户端等待没有必要等待的东西.

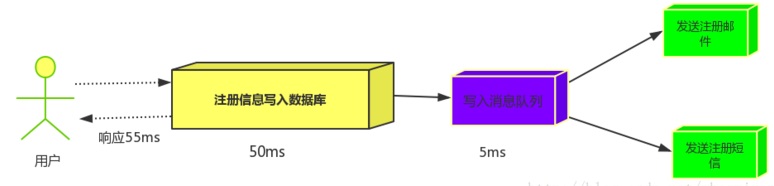


1. 并行方式:将注册信息写入数据库后,发送邮件的同时,发送短信,以上三个任务完成后,返回给客户端,并行的方式能提高处理的时间。



假设三个业务节点分别使用50ms,串行方式使用时间150ms,并行使用时间100ms。虽然并性已经提高的处理时间,但是,前面说过,邮件和短信对我正常的使用网站没有任何影响，客户端没有必要等着其发送完成才显示注册成功,应该是写入数据库后就返回.

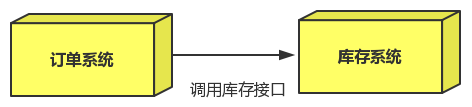
1. 消息队列 ：引入消息队列后，把发送邮件,短信不是必须的业务逻辑异步处理。



由此可以看出,引入消息队列后，用户的响应时间就等于**写入数据库的时间+写入消息队列的时间**(可以忽略不计),引入消息队列后处理后,响应时间是串行的3倍,是并行的2倍。

## 应用解耦

场景：双11是购物狂节,用户下单后,订单系统需要通知库存系统,传统的做法就是订单系统调用库存系统的接口.

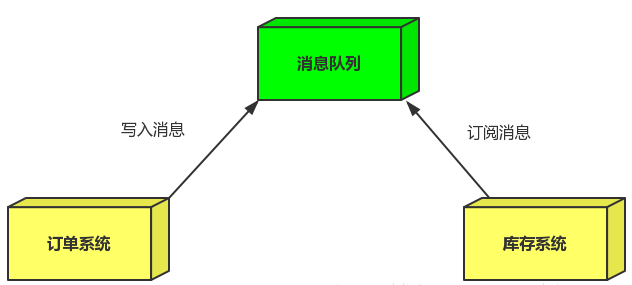


这种做法有一个缺点:

当库存系统出现故障时,订单就会失败。(这样马云将少赚好多好多钱^ ^)

订单系统和库存系统高耦合.

引入**消息队列**



**订单系统**:用户下单后,订单系统完成**持久化处理**,将消息写入消息队列,返回用户订单下单成功。

**库存系统**:订阅下单的消息,获取下单消息,进行库操作。

就算库存系统出现故障,消息队列也能保证消息的可靠投递,不会导致消息丢失(马云这下高兴了).

## 流量削峰

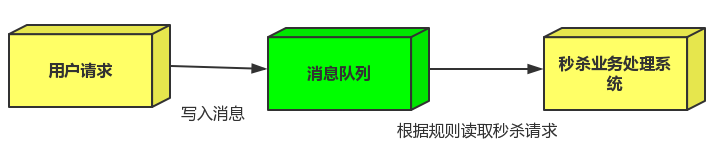
**流量削峰**一般在**秒杀**活动中应用广泛

场景:秒杀活动，一般会因为流量过大，导致应用挂掉,为了解决这个问题，一般在**应用前端加入消息队列**。

作用:

1.可以控制活动人数，超过此一定阀值的订单直接丢弃(我为什么秒杀一次都没有成功过呢^^)

2.可以缓解短时间的高流量压垮应用(应用程序按自己的最大处理能力获取订单)



**用户的请求,**服务器收到之后,首先写入消息队列,加入消息队列长度超过最大值,则直接抛弃用户请求或跳转到错误页面；秒杀业务根据消息队列中的请求信息，再做后续处理。

## 具体描述场景

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。

场景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的基本概念

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。

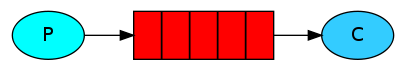
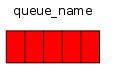
### **Connection**是RabbitMQ的**socket链接**，它封装了socket协议相关部分逻辑。

### **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。

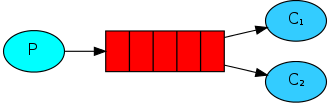
## Queue

Queue（队列）是RabbitMQ的**内部对象**，用于**存储消息**，用下图表示。



RabbitMQ中的消息都只能存储在***Queue***中，生产者Producer（下图中的P）生产消息并最终投递到Queue中，消费者Consumer（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消息确认

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…

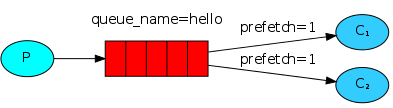
另外pub message是没有ack的。

## Message durability消息持久化

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。

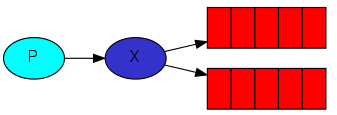
## Prefetch count

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。



## Exchange交换机

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）**。



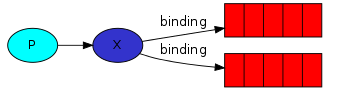
Exchange是按照什么逻辑将消息路由到Queue的？这个将在Binding一节介绍。 RabbitMQ中的Exchange有四种类型，不同的类型有着不同的路由策略，这将在Exchange Types一节介绍。

## 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交换机类型

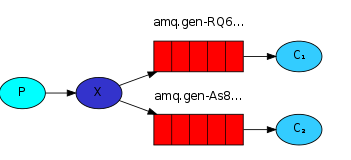
RabbitMQ常用的Exchange Type有**fanout、direct、topic、headers**这四种（AMQP规范里还提到两种Exchange Type，分别为system与自定义，这里不予以描述），下面分别进行介绍。

## fanout

**Fanout Exchange:广播订阅**,向所有的消费者发布消息,但是只有消费者将队列绑定到该路由器才能收到消息,忽略Routing Key.

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。

fanout 英 [fænaʊt] 美 [fænaʊt] [经] 分列（帐户）

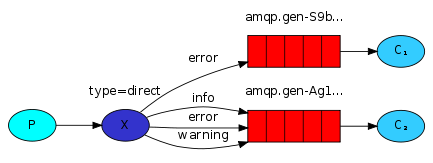


上图中，生产者（P）发送到Exchange（X）的所有消息都会路由到图中的两个Queue，并最终被两个消费者（C1与C2）消费。

## direct

**Direct Exchange**:直接匹配,通过**Exchange名称+RountingKey**来发送与接收消息.

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**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

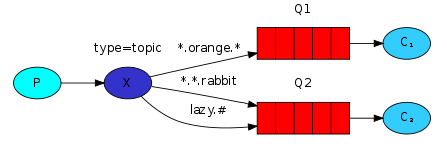
**Topic Exchange**：**主题匹配订阅**,这里的主题指的是RoutingKey,RoutingKey可以采用通配符,如:\*或#，RoutingKey命名采用.来分隔多个词,只有消息这将队列绑定到该路由器且指定RoutingKey符合匹配规则时才能收到消息;

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。

headers Exchange:消息头订阅,消息发布前,为消息定义一个或多个键值对的消息头,然后消费者接收消息同时需要定义类似的键值对请求头:(如:x-mactch=all或者x\_match=any)，只有请求头与消息头匹配,才能接收消息,忽略RoutingKey.

**默认的exchange**:如果用**空字符串**去声明一个exchange，那么系统就会使用**”amq.direct”**这个exchange，我们创建一个queue时,默认的都会有一个和新建queue同名的**routingKey**绑定到这个默认的exchange上去。

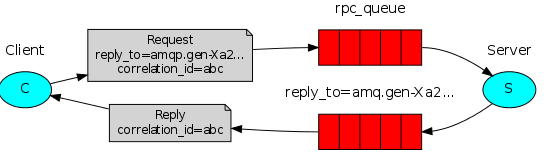
channel.BasicPublish("", "TaskQueue", properties, bytes);

因为在第一个参数选择了**默认的exchange**，而我们申明的队列叫**TaskQueue**，所以默认的，它在新建一个也叫**TaskQueue的routingKey**，并绑定在默认的exchange上，导致了我们可以在第二个参数routingKey中写TaskQueue，这样它就会找到定义的同名的queue，并把消息放进去。

如果有两个接收程序都是用了同一个的queue和相同的routingKey去绑定direct exchange的话，分发的行为是负载均衡的，也就是说第一个是程序1收到，第二个是程序2收到，以此类推。 如果有两个接收程序用了各自的queue，但使用相同的routingKey去绑定direct exchange的话，分发的行为是复制的，也就是说每个程序都会收到这个消息的副本。行为相当于fanout类型的exchange。

# 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。



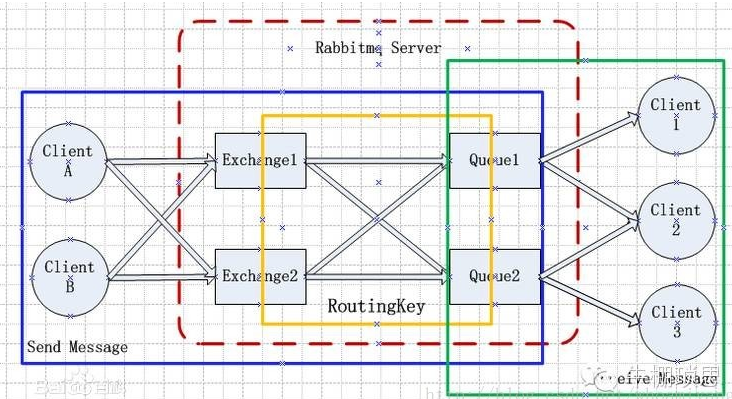
RabbitMQ 中实现RPC 的机制是：

客户端发送请求（消息）时，在消息的属性（**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。

# 系统架构



几个概念说明:

Broker:它提供一种**传输服务**,它的角色就是**维护一条从生产者到消费者的路线**，保证数据能按照指定的方式进行传输,

**Exchange**：消息交换机,它指定消息按什么规则,路由到哪个队列。

Queue:消息的载体,每个消息都会被投到一个或多个队列。

**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里可以有多个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。

# 任务分发机制

## Round-robin dispathching循环分发

**RabbbitMQ**的**分发机制**非常适合扩展,而且它是专门为**并发程序**设计的,如果现在load加重,那么只需要创建更多的Consumer来进行任务处理。

## Message acknowledgment消息确认

**为了保证数据不被丢失,RabbitMQ支持消息确认机制**,为了保证数据能被正确处理而不仅仅是被Consumer收到,那么我们不能采用no-ack，而应该是在处理完数据之后发送ack.

在处理完数据之后发送ack,就是告诉RabbitMQ数据已经被接收,处理完成,RabbitMQ可以安全的删除它了.

如果Consumer退出了但是没有发送ack,那么RabbitMQ就会把这个Message发送到下一个Consumer，这样就保证在Consumer异常退出情况下数据也不会丢失.

**RabbitMQ它没有用到超时机制**。RabbitMQ仅仅通过Consumer的连接中断来确认该Message并没有正确处理，也就是说RabbitMQ给了Consumer足够长的时间做数据处理。

如果忘记ack,那么当Consumer退出时,Mesage会重新分发,然后RabbitMQ会占用越来越多的内存.

# Message durability消息持久化

要持久化队列queue的持久化需要在声明时指定**durable=True**;

这里要注意,队列的名字一定要是**Broker**中不存在的,不然不能改变此队列的任何属性.

队列和交换机有一个创建时候指定的标志durable,**durable的唯一含义就是具有这个标志的队列和交换机会在重启之后重新建立,**它不表示说在队列中的消息会在重启后恢复。

消息持久化包括3部分：

### **exchange持久化**,在声明时指定durable => true

hannel.ExchangeDeclare(ExchangeName, "direct", **durable: true**, autoDelete: false, arguments: null);//声明消息队列，且为可持久化的

### **queue持久化**,在声明时指定durable => true

channel.QueueDeclare(QueueName, **durable: true**, exclusive: false, autoDelete: false, arguments: null);//声明消息队列，且为可持久化的

### **消息持久化**,在投递时指定**delivery\_mode => 2**(1是非持久化).

channel.**basicPublish**("", queueName, MessageProperties.PERSISTENT\_TEXT\_PLAIN, msg.getBytes());

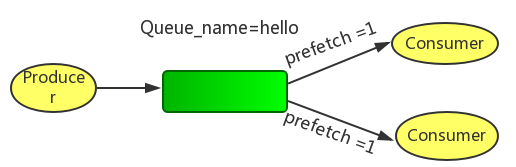
**如果exchange和queue都是持久化的,那么它们之间的binding也是持久化的,如果exchange和queue两者之间有一个持久化，一个非持久化,则不允许建立绑定.**

注意：一旦创建了队列和交换机,就不能修改其标志了,例如,创建了一个non-durable的队列,然后想把它改变成durable的,唯一的办法就是删除这个队列然后重现创建。

# Fair dispath 公平分发

你可能也注意到了,分发机制不是那么优雅,默认状态下,RabbitMQ将第n个Message分发给第n个Consumer。n是取余后的,它不管Consumer是否还有unacked Message，只是按照这个默认的机制进行分发.

那么如果有个Consumer工作比较重,那么就会导致有的Consumer基本没事可做,有的Consumer却毫无休息的机会,那么,Rabbit是如何处理这种问题呢?



通过basic.qos方法设置prefetch\_count=1，这样RabbitMQ就会使得每个Consumer在同一个时间点最多处理一个Message，换句话说,在接收到该Consumer的ack前,它不会将新的Message分发给它

channel.basic\_qos(prefetch\_count=1)

注意，这种方法可能会导致queue满。当然，这种情况下你可能需要添加更多的Consumer，或者创建更多的virtualHost来细化你的设计。

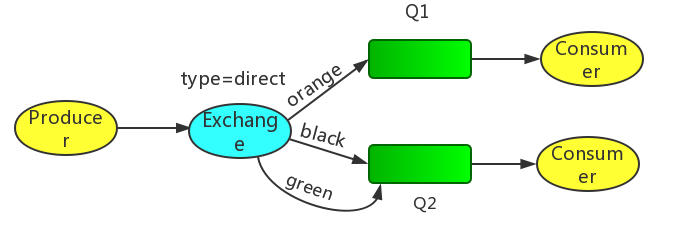
# 分发到多个Consumer

## Bindings 绑定

绑定其实就是关联了**exchange和queue**，或者这么说:queue对exchange的内容感兴趣,exchange要把它的Message deliver到queue。

## Direct exchange

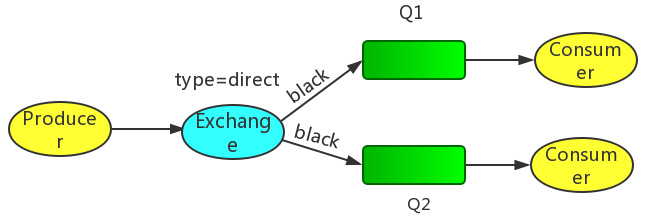
Driect exchange的路由算法非常简单:**通过bindingkey的完全匹配**，可以用下图来说明.



Exchange和两个队列绑定在一起,Q1的bindingkey是orange，Q2的binding key是black和green. 当Producer publish key是orange时,exchange会把它放到Q1上,如果是black或green就会到Q2上,其余的Message被丢弃.

## Multiple bindings

多个queue绑定同一个key也是可以的,对于下图的例子,Q1和Q2都绑定了black,对于routing key是**black**的Message，会被deliver到Q1和Q2，其余的Message都会被丢弃.

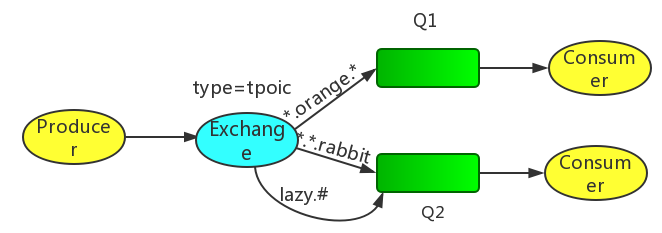


## Topic exchange

对于Message的routing\_key是有限制的，不能使任意的。**格式是以点号“.”分割的字符表。**比如：”stock.usd.nyse”, “nyse.vmw”, “quick.orange.rabbit”。你可以放任意的key在routing\_key中，当然最长不能超过255 bytes。

对于routing\_key，有两个特殊字符：

**\*(星号)代表任意一个单词；#(hash)0个或多个单词**



Producer发送消息时需要设置**routing\_key**，routing\_key包含三个单词和连个点号o,第一个key描述了**celerity**(灵巧),第二个是**color**(色彩),第三个是**物种**:

在这里我们创建了两个绑定： Q1 的binding key 是”.orange.“； Q2 是 “..rabbit” 和 “lazy.#”：

Q1感兴趣所有orange颜色的动物

Q2感兴趣所有rabbits和所有的lazy的.

例子:rounting\_key 为 “quick.orange.rabbit”将会发送到Q1和Q2中

rounting\_key 为”lazy.orange.rabbit.hujj.ddd”会被投递到Q2中,#匹配0个或多个单词。

# 消息序列化

**RabbitMQ**使用**ProtoBuf序列化消息**,它可作为RabbitMQ的Message的数据格式进行传输,由于是**结构化的数据**,这样就极大的方便了Consumer的数据高效处理,当然也可以使用XML，与XML相比,ProtoBuf有以下优势:

1.简单

2.size小了3-10倍

3.速度快了20-100倍

4.易于编程

6.减少了语义的歧义.

ProtoBuf具有速度和空间的优势，使得它现在应用非常广泛。

# RabbitMQ、Kafka、activeMQ、ZeroMQ对比

RabbitMQ 选型和对比

### 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。

### 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**。

### 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**。

### 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。

### 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 胜于 Kafka**（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**。