# RabbitMQ学习指南

|  |  |  |  |
| --- | --- | --- | --- |
| 版本 | 日期 | 作者 | 备注 |
| V1 | 2019-08-20 | 谭祖好 | 1. 安装文档..\RabbitMQ\安装文档 2. 项目初版springboot-V3 3. 项目进阶版springboot-V4 |

## RabbitMQ简介

RabbitMQ是一个开源的消息代理和队列服务器，用来通过普通协议在完全不同的应用之间共享数据，RabbitMQ是使用Erlang语言来编写的，并且RabbitMQ是基于 AMQP协议的。（备注：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。）

## RabbitMQ特点

### 2.1 RabbitMQ底层采用Erlang语言进行编写。

服务端用Erlang语言编写（Erlang是一种通用的面向[并发](https://baike.baidu.com/item/%E5%B9%B6%E5%8F%91)的编程语言）。

### 2.2 开源、性能优秀，稳定。

一个开源的AMQP实现

### 2.3与SpringAMQP完美的整合、API丰富。

### 2.4集群模式丰富，表达式配置，HA模式，镜像队列模型

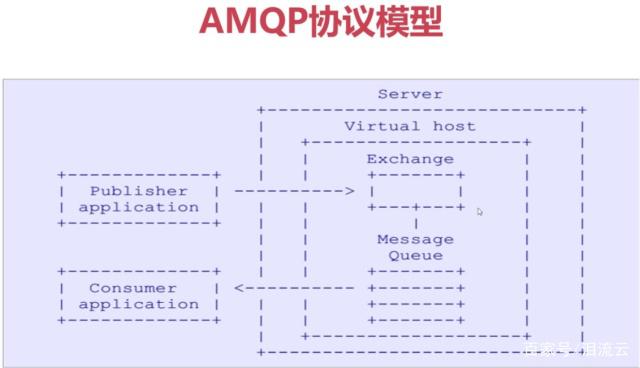
### 2.5保证数据不丢失的前提做到高可靠、可用性。

消息持久化：从内存持久化消息到硬盘,再从硬盘加载到内存。

## AMQP协议模型

AMQP: 全称Advanced Message Queuing Protocol （高级消息队列协议）。

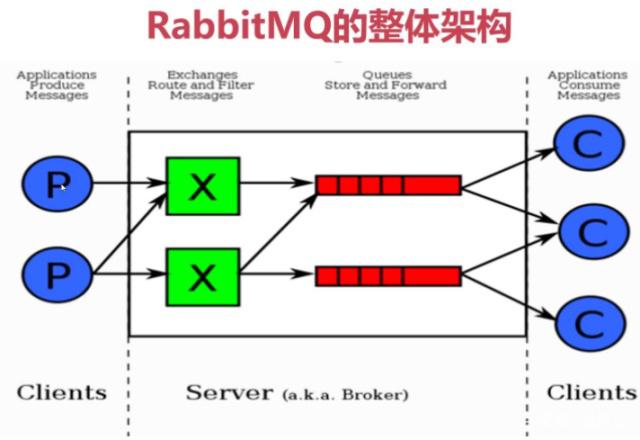
是应用层协议的一个开放标准，为面向消息的中间件设计。



## RabbitMQ安装和使用

详见附件安装文档，路径：附件\RabbitMQ\安装文档

## RabbitMQ核心概念



P:即producer（生产者），X:即Exchange(交换器) ,C:即Consumer（消费者）。

### 5.1 Broker

又称Server(实体服务)，接收客户端的连接，实现AMQP实体服务。它提供一种传输服务，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。

### 5.2 Exchange

交换机，接收消息，根据路由键转发消息到绑定的队列。

### 5.3 Queue

即Message Queue，消息队列。保存消息并将它们转发给消费者，消费者直接监听队列就能收到消息。

### 5.4 Binding

绑定。作用：把Exchange和Queue按照路由规则绑定起来。

绑定的时候一般会指定一个绑定键( BindingKey ) ，这样 RabbitMQ 就知道如何正确地将消息路由到队列。

### 5.5 Routing Key

路由关键字(路由规则)。Exchange根据这个路由规则进行消息投递。

### 5.6 Virtual Host

虚拟地址，用于进行逻辑隔离，最上层的消息路由。一个Virtual host里面可以有若干个Exchange和Queue，同一个Virtual host里面不能有相同名称的Exchange或Queue（用于隔离不同的项目和应用的）

### 5.7 Message

消息。服务器和应用程序之间传送的数据。

5.8 Connection

连接，应用程序与Broker的网络连接

### 5.9 Channel

消息通道(消息读写的通道)，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。Channel是与RabbitMQ打交道的最重要的一个接口。

### 5.10 Producer

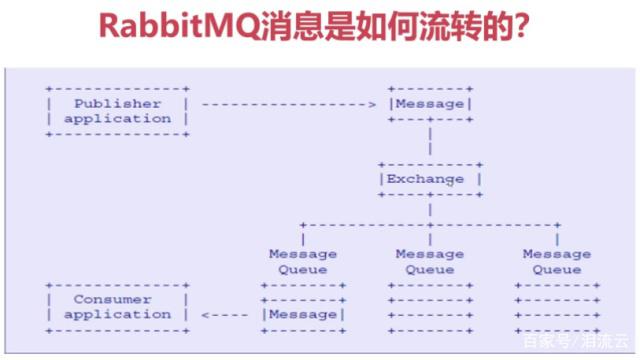
消息生产者，投递消息的程序。

### 5.11 Consumer

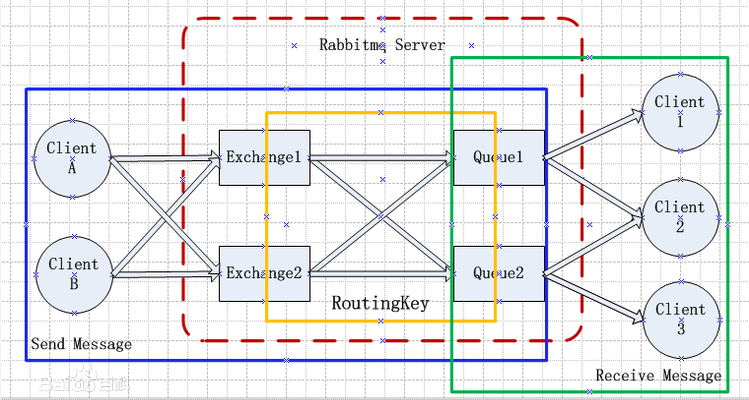
消息消费者,接收消息的程序。

## RabbitMQ消息流转

简单流程：



复杂流程图:



生产者将消息发送到Exchange(交换机),由Exchange将消息路由到一个或者多个Queue中，消费者可以从Queue中获取消息并消费（当多个消费者订阅同一个Queue时，Queue中的消息会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）。

## Exchange 4种模式

Exchange按照特定的模式转发消息到相应的Queue进行存储，

常见的模式有4种。

### 7.1 Direct模式

直接模式(RabbitMQ默认模式)，又称“点对点模式”。所有发送到Direct Exchange的消息，会被转发到RouteKey中指定的Queue（RouteKey可以简单理解为要发送到的队列名字）。

#### 7.1.1 消息发给唯一一个节点使用

#### 7.1.2 可以使用rabbitMQ自带的Exchange(default Exchange)。

#### 7.1.3 Direct类型下不需要将Exchange进行任何绑定(binding)操作

#### 7.1.4 RouteKey和队列名要匹配。

生产者发消息时，消息指定的路由关键字(RouteKey)，在虚拟主机(vhost)中根据RouteKey找不到队列名，则该消息会被抛弃。

### 7.2 Topic模式

主题模式，Exchange根据路由关键字，将消息按照匹配规则，路由到一个或者多个Queue中（或者丢弃）。

#### 7.2.1 和Direct模式区别

top1.在匹配规则上进行了扩展

top2.可以发给多个节点(队列)使用

#### 7.2.3 匹配规则

top1. RoutingKey(路由键)用"."分割的字母或数字

top2. \* 表示匹配单个字母或数字，# 表示匹配0~多个字母或数字。

top3. 例子：

“\*.\*.a.#”表示：

"任意1个字母或数字. 任意1个字母或数字.a. n个字母或数字"。其中 n>=0。

### 7.3. Fanout模式

广播模式，把消息路由到与该Exchange绑定所有的队列中。

#### 7.3.1 给Exchange发送消息，绑定了这个Exchange的所有队列都收到这个消息。

#### 7.3.2 不需要处理RouteKey,只需要简单的将队列绑定到Exchange上。

#### 7.3.3 发送到Exchange的消息，都会被转发到与该Exchange绑定的所有队列上。

#### 7.3.4 Fanout类型 Exchange 转发消息是最快的。

### 7.4 Header 模式

头部类型，headers类型的Exchange(交换器)性能很差,所以现实中使用较少。

## 相关注解

### 8.1 @Payload

可以接受对象型消息。此注解加上之后，spring会把message格式的数据，自动转换为此注解后面跟着的对象。

### 8.2 @RabbitListener

指定目标方法来作为消费消息的方法，通过注解参数指定所监听的队列或者Binding。

@RabbitListener//消息监听器

(

bindings = @QueueBinding//将交换机和队列绑定

(value = @Queue(name ="队列名", durable = "true"//队列持久化),

exchange //声明交换机

= @Exchange(name = "交换机名", type = ExchangeTypes.TOPIC,//交换机类型

ignoreDeclarationExceptions = "true"//忽略声明异常, durable = "true"//交换机持久化),

key = {"","" }//返回绑定的路由密钥(route key)，多个路由密钥将导致多个绑定。

)

## 9.消息回调

确认生产者发送消息成功。只确认消息是否正确到达 Exchange 中，消费者是否处理成功不做保证。

## 10.消息确认(message acknowledgment)

### 10.1 消息确认(ack机制)

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。

### 10.2 注意事项

在消费者处理完业务逻辑后，忘记发送回执给RabbitMQ后果:

#### 9.2.1 队列中堆积的消息会越来越多。

#### 9.2.2 服务重启后，消费者会重复消费这些消息并重复执行业务逻辑。

核心代码: channel.basicAck(deTag, false);// false:手动确认

## 11.可持久化

RabbitMQ支持消息的持久化，也就是数据写在磁盘上，为了数据安全考虑，应该选择持久化。

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，但它不表示说在队列当中的消息会在重启后恢复。例如：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，就不允许建立绑定。

**注意**：一旦创建了队列和交换机，就不能修改其标志了。例如：如果创建了一个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**。

### 11.1. 交换机持久化

#### 11.1.1 在服务端声明queue时，进行交换机持久化

new TopicExchange(String name);//默认持久化，即durable为true

#### 11.1.2 在客户端声明exchange时，进行交换机持久化

@RabbitListener//消息监听器

(

bindings = exchange //声明交换机

= @Exchange(name = "交换机名", type = ExchangeTypes.TOPIC,//交换机类型

ignoreDeclarationExceptions = "true"//忽略声明异常, durable = "true"//交换机持久化),

……….

)

### 11.2 队列持久化

#### 11.2.1. 在服务端声明queue时，进行队列持久化。

new Queue(String name, boolean durable);// durable为true表示队列持久化。

#### 11.2.2. 客户端声明queue时，也要队列持久化

@RabbitListener//消息监听器

(

bindings = @QueueBinding//将交换机和队列绑定

(value = @Queue(name ="队列名", durable = "true"//队列持久化),

…...

)

### 11.3 消息可持久化（message durability）

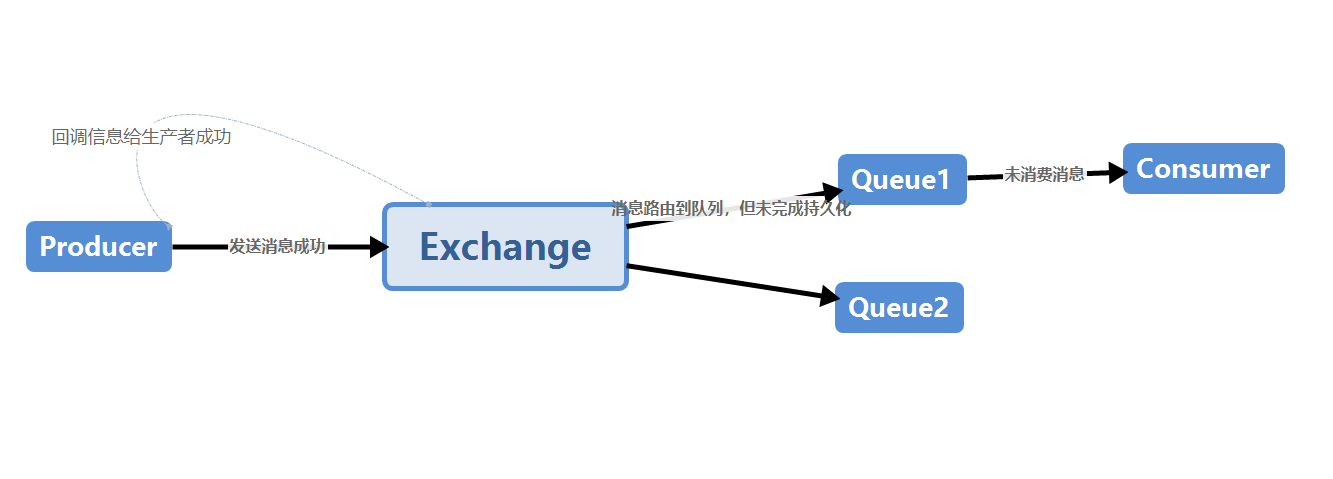
#### 11.3.1.消息持久化

RabbitMQ服务挂掉或者重启的情况下，消息会出现丢失问题，所以需要将消息存储起来，可以保证绝大部分情况下不丢失。

生产者发送消息和消费端接收消息时，默认DeliveryMode为true，只要成功持久化交换器和队列，不需要任何设置，就能成功消息持久化。

#### 11.3.2 消息小概率丢失事件

RabbitMQ服务器已经接收到生产者的消息，但还没有来得及持久化该消息时，RabbitMQ服务器就挂掉。



#### 11.2.3 消息小概率丢失事件解决方案

可能引起消息丢失的异常：消费端取到消息后进程退出，重启。

ack机制，设置autoAck=False,关闭自动ack, 真正处理完成后，手动发送完成确认。如果没有ack确认 且当前消费者链接断开，任务会重新进入队列。

## 12 控制消费者流量（prefetch count）。

### 12.1.设置prefecth count来控制消费者流量

设置prefetchCount可以限制Queue每次发送给每个消费者的消息数。

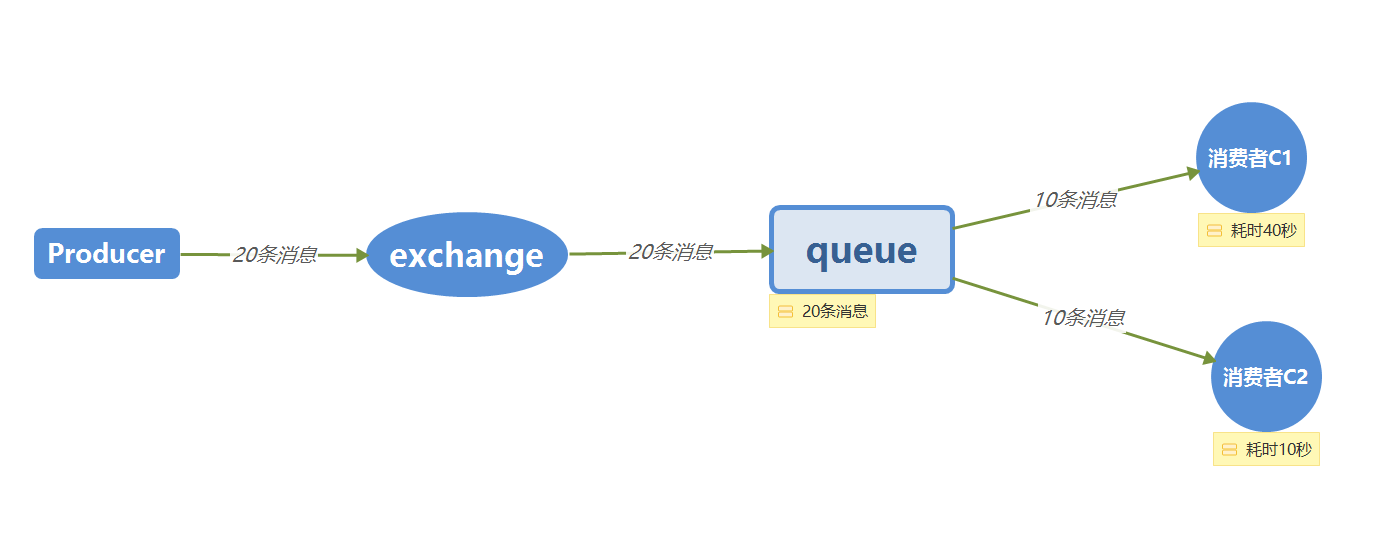
例如:设置prefetchCount=1，则Queue每次给每个消费者发送一条消息；

消费者处理完这条消息，Queue会再给该消费者发送一条消息, 如果消费者中有一条消息没有处理完，Queue就不会继续给这个消费者发送新消息。

### 12.2 设置prefecth count的原因

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。

如图所示，完成整个工作耗时40秒，其中T1=40秒 , T2=10秒，C2闲了30秒。



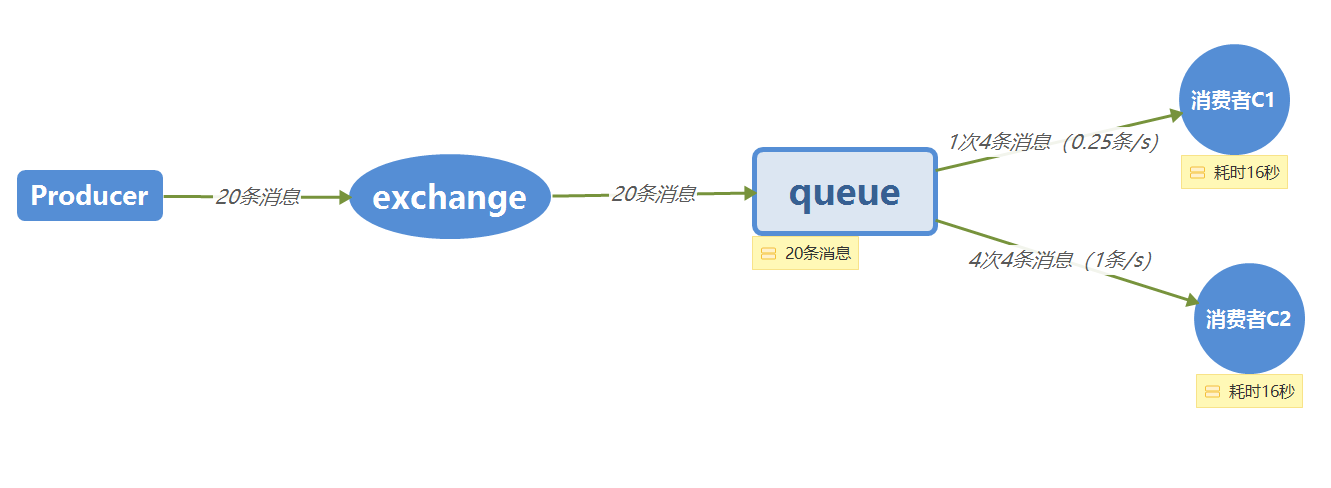
忽略queue耗时粗算: V1=10条/40s=0.25条/s ，V2=10条/10s=1条/s

设共同耗时为t，方程:0.25t + t= 2 0 条, 解得：t=16s

消息理论分配：C1分得4条消息(16\*0.25) , C2分得16条消息(1\*16)

可设置prefecthCount=4, Queue每次给C1、C2发送4条消息。

如图，完成整个工作耗时16秒，T1=T2=16秒，这期间C1、C2都没有闲着。



### 12.3 合理设置prefetch count值

#### 12.3.1 设置不合理

top1.值过大。

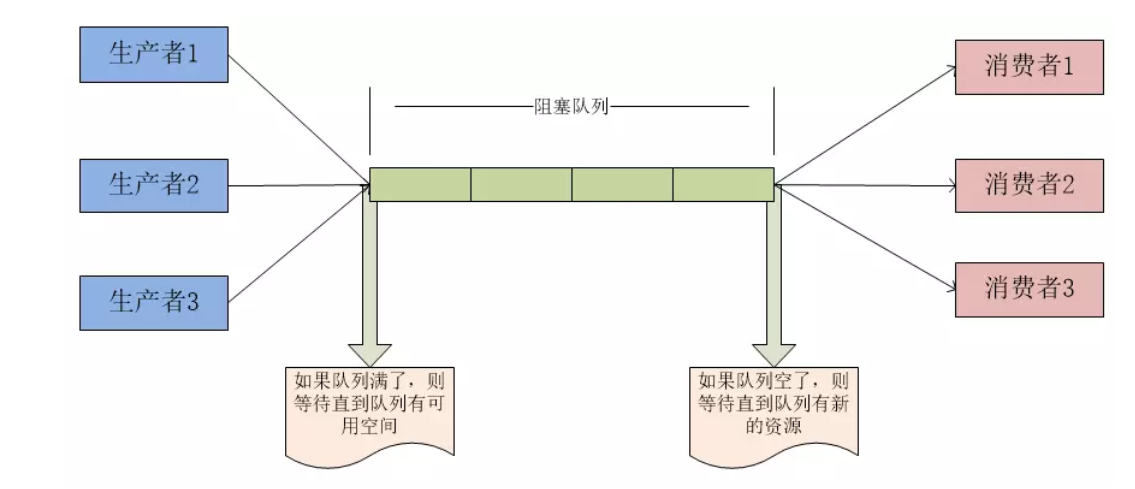
可能导致消费者处理不过来，一直在本地缓存的BlockingQueue(阻塞队列)里呆太久，这样消息在客户端的延迟就大大增加；

而对于多个消费者的情况，则会分配不均匀，导致有些消费者一直在忙，有些则非常空闲。

top2.值过小

令到消费者不能充分工作，因为我们总想它100％的时间都是处于繁忙状态，而这时可能会在处理完一条消息后，BlockingQueue为空，因为新的消息还未来得及到达，所以消费者就处于空闲状态了。

BlockingQueue（阻塞队列）如图所示：



#### 合理设置考虑因素

top1.客户端服务端之间网络传输时间。

top2.消费者消耗一条消息所执行的业务逻辑的耗时。

top3.网络状况。

## 13.消息100%投递成功设计方案

### 13.1可能丢失消息的异常

异常情况：服务异常退出或重启后，exchange，queue，message丢失。

#### 13.1.1.生产者

top1. 消息发送后没有投放到指定队列

top2 消息routing\_key不存在

#### 13.1.2消费者

消费端取到消息后，还没有进行持久化，进程退出或服务重启。

### 13.2.投递方案

#### 13.2.1数据库表设计

1、消息信息表（即业务数据表）msg

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 列名称 | 列类型 | 长度 | 约束 | 备注 |
| id | int | 11 | primary key | 主键ID |
| param …. | …… | …… | ……… | 消息业务参数 |

2、消息状态表msg\_status

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 列名称 | 列类型 | 长度 | 约束 | 备注 |
| id | int | 11 | primary key | 主键ID |
| status | int | 1 |  | 消息状态（0:未发送,1:回调成功,2.回调失败,3.确认成功,4.确认失败） |
| msg\_id | int | 11 |  | 关联消息信息表主键ID |
| retry\_count | int | 2 |  | 重复发送/处理次数 |
| param …. | …… | …… | ……… | 其他参数 |

#### 13.2.2 步骤

Step 1 将要发送的消息存储到数据库表,即msg表和msg\_status 表，其中msg\_status表的消息状态为“未发送”(status=0)。

Step 2 创建交换机、队列时，进行交换机持久化(默认)、队列持久化(durable=true)。

Step3发送消息到MQ Broker节点,采用confirm方式发送,会有异步的返回结果。

Step4 生产者端接受MQ Broker节点返回的Confirm确认消息结果，然后进行更新消息状态表(msg\_status)的消息状态(回调成功:status=1,回调失败:status=2)。

Step 4 消费进行交换机和队列绑定时，持久化交换机和队列(durable=true)、消息持久化（默认）。

Step5消费端采用ack机制进行手动确认。出现异常，消息重新入队列，轮询3次仍然失败，消息拒绝入队列，记录到消息状态表(msg\_status)。

Step5对于发送失败的消息，或定时器重发，或登录RabbitMQ后台手工处理，或不处理(消息要求随机或实时则不需做处理)。

## 14.RabbitMQ应用场景

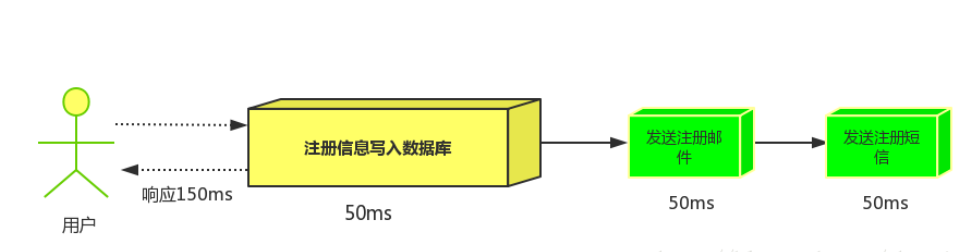
应用场景参考：<https://blog.csdn.net/qq_14901335/article/details/80451052>

### 14.1 异步处理

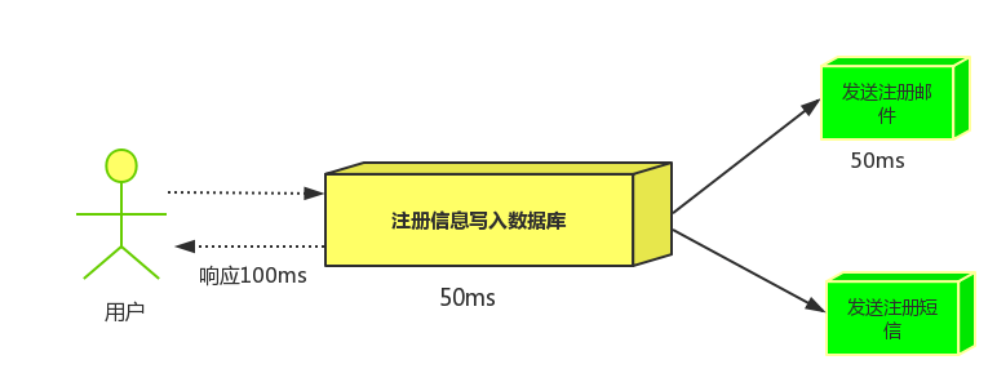
场景说明：用户注册后，需要发注册邮件和注册短信,传统的做法有两种

1.串行的方式 、2.并行的方式

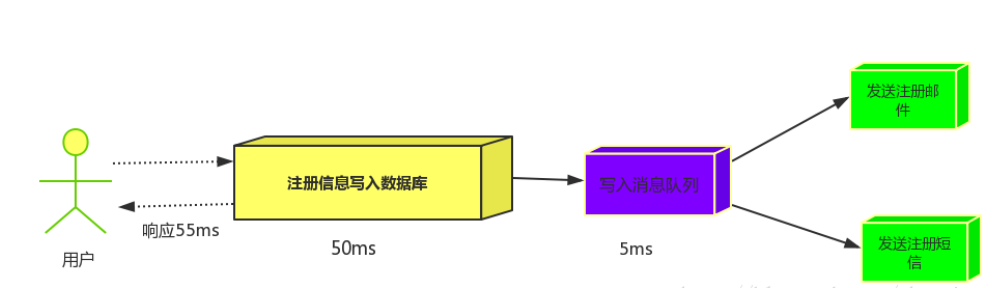
(1)串行方式:将注册信息写入数据库后,发送注册邮件,再发送注册短信,以上三个任务全部完成后才返回给客户端。邮件,短信并不是必须的,它只是一个通知,而这种做法让客户端等待没有必要等待的东西。



(2)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。



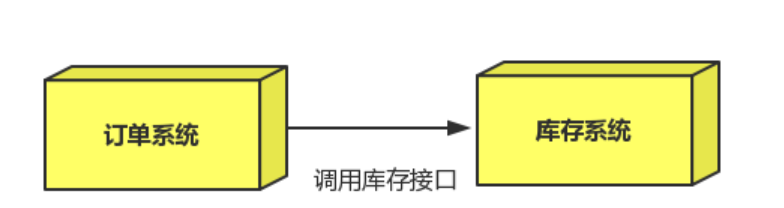
(3)消息队列   
引入消息队列后，把发送邮件,短信不是必须的业务逻辑，进行异步处理 。



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。

### 14.2 应用解耦

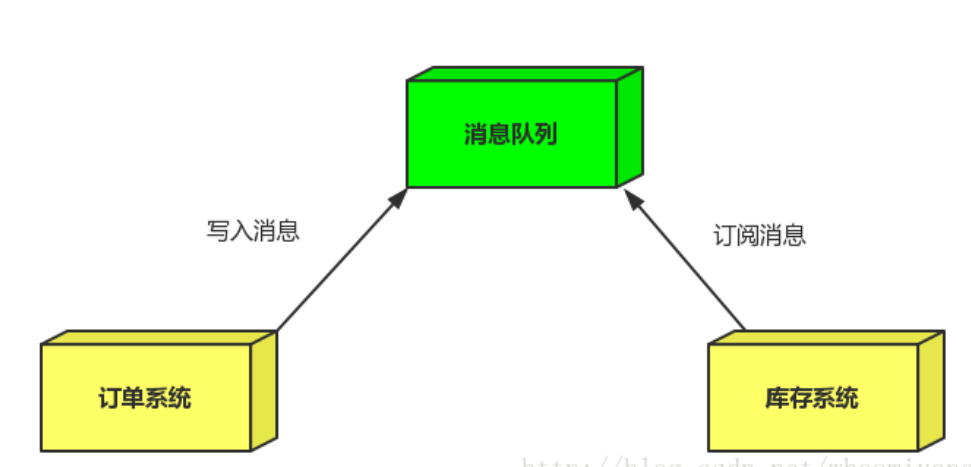
场景：双11是购物狂节,用户下单后,订单系统需要通知库存系统,传统的做法就是订单系统调用库存系统的接口.



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。
* 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。

### 14.3 流量削峰

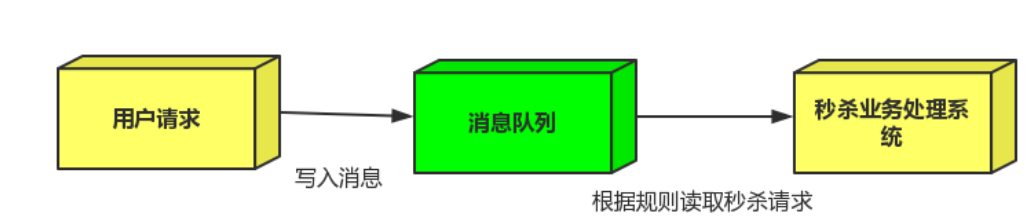
流量削峰，一般在秒杀活动中应用广泛 。

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。

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) 。



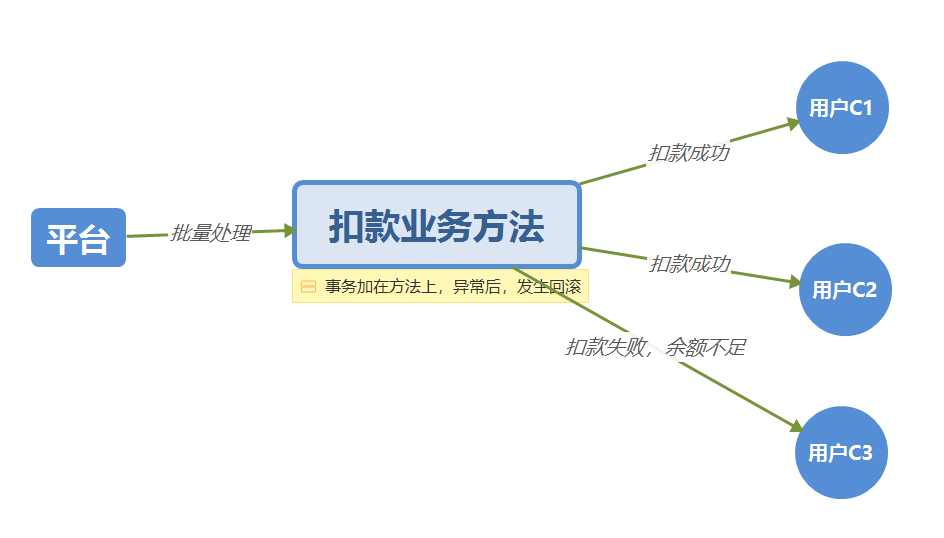
1.用户的请求,服务器收到之后,首先写入消息队列,加入消息队列长度超过最大值,则直接抛弃用户请求或跳转到错误页面。  
2.秒杀业务根据消息队列中的请求信息，再做后续处理。

### 14.4 工作案例-平台对用户银行卡进行扣款。

案例描述：借款用户授权给平台开通银行自动扣款业务后，到还款当日，平台批量自动从借款用户银行卡扣费。

其中，用户银行卡余额，平台无权限查询，平台可调用扣款接口，余额充足，则扣款成功，反之失败。

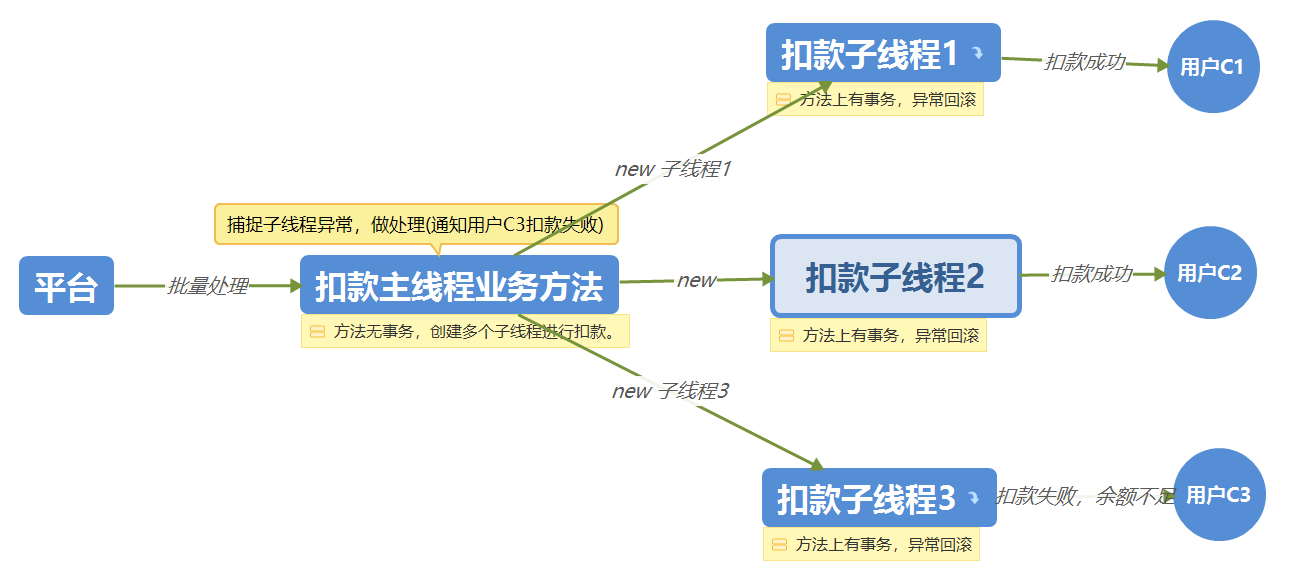
#### 14.4.1 串行处理



串行缺点：

1. 整个流程都在主线程，主线程的请求耗时可能会非常长，如果请求过多，出现排队请求，甚至宕机，严重影响性能。
2. 事务加在方法上，一旦有用户扣款失败，将全部回滚。
3. 用户做没有必要等待。

#### 14.4.2 并行处理



缺点：性能差。

由于异步线程里的操作都是很耗时间的操作，一个请求要开启2个线程，而一台标准配置的ECS服务器支撑的并发线程数大概在800左右，假设一个线程在10秒做完，这个单个服务器最多能支持400个请求的并发，后面的就要排队。出现这种情况，就要考虑增加服务器做负载，尴尬的增加成本。

#### 14.4.3 消息队列处理

