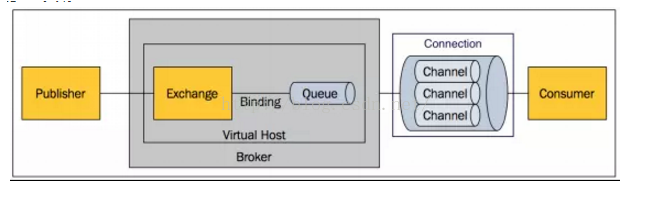
# rabbitMQ的基础知识

## 1.什么是MQ？

       MQ全称为Message Queue, 消息队列（MQ）是一种应用程序对应用程序的通信方法。MQ是消费-生产者模型的一个典型的代表，一端往消息队列中不断写入消息，而另一端则可以读取队列中的消息。

rabbitMQ的内部结构如下：



### 相关概念：

**Message**  
消息，消息是不具名的，它由消息头和消息体组成。消息体是不透明的，而消息头则由一系列的可选属性组成，这些属性包括routing-key（路由键）、priority（相对于其他消息的优先权）、delivery-mode（指出该消息可能需要持久性存储）等。

**Publisher**  
消息的生产者，也是一个向交换器发布消息的客户端应用程序。

**Exchange**  
交换器，用来接收生产者发送的消息并将这些消息路由给服务器中的队列。

**Binding**  
绑定，用于消息队列和交换器之间的关联。一个绑定就是基于路由键将交换器和消息队列连接起来的路由规则，所以可以将交换器理解成一个由绑定构成的路由表。

**Queue**  
消息队列，用来保存消息直到发送给消费者。它是消息的容器，也是消息的终点。一个消息可投入一个或多个队列。消息一直在队列里面，等待消费者连接到这个队列将其取走。

**Connection**  
网络连接，比如一个TCP连接。

**Channel**  
信道，**多路复用**连接中的一条独立的双向数据流通道。信道是建立在真实的**TCP连接内的虚拟连接**，AMQP 命令都是通过信道发出去的，不管是发布消息、订阅队列还是接收消息，这些动作都是通过信道完成。因为对于操作系统来说建立和销毁 TCP 都是非常昂贵的开销，所以引入了信道的概念，以复用一条 TCP 连接。

**Consumer**  
消息的消费者，表示一个从消息队列中取得消息的客户端应用程序。

**Virtual** **Host**  
虚拟主机，表示一批交换器、消息队列和相关对象。虚拟主机是共享相同的身份认证和加密环境的独立服务器域。每个 vhost 本质上就是一个 mini 版的 RabbitMQ 服务器，拥有自己的队列、交换器、绑定和权限机制。vhost 是 AMQP 概念的基础，必须在连接时指定，RabbitMQ 默认的 vhost 是 / 。

**Broker**  
表示消息队列服务器实体。

RabbitMQ 是用 Erlang 语言写的。

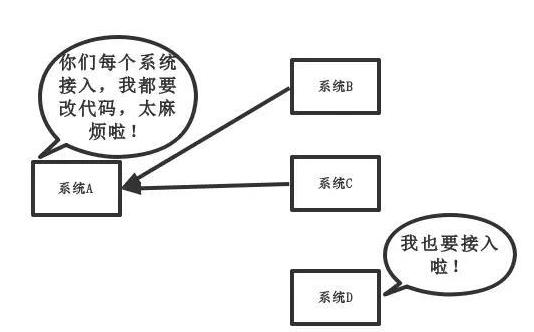
# 消息队列的优缺点

## 问题：为什么要使用消息队列？优点？

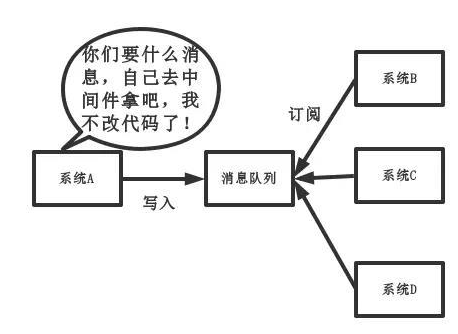
主要的有三个：解耦、异步和削峰。

### 解耦：

传统模式：系统间耦合性太强，如下图所示，系统A在代码中直接调用系统B和系统C的代码，如果将来D系统接入，系统A还需要修改代码，过于麻烦！即一个系统或者一个模块，调用了多个系统或者模块，互相之间的调用很复杂，维护起来很麻烦。但是其实这个调用是不需要直接同步调用接口的，如果用MQ给他异步化解耦，也是可以的.



加入消息队列以后的中间件模式：

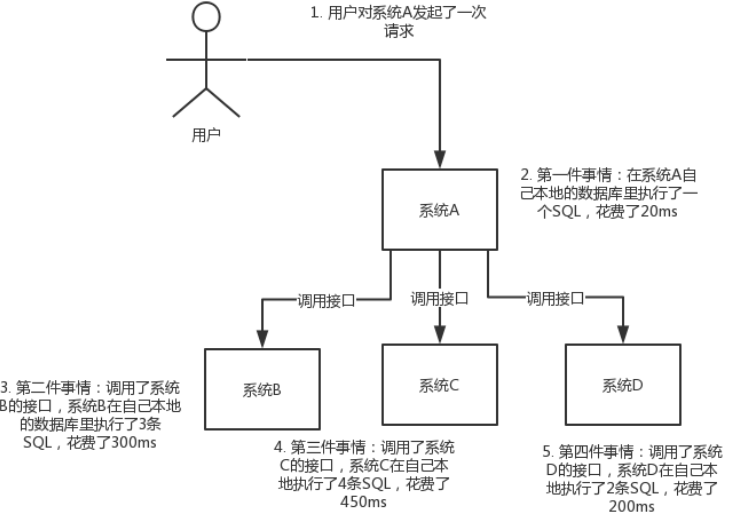


中间件模式的优点：

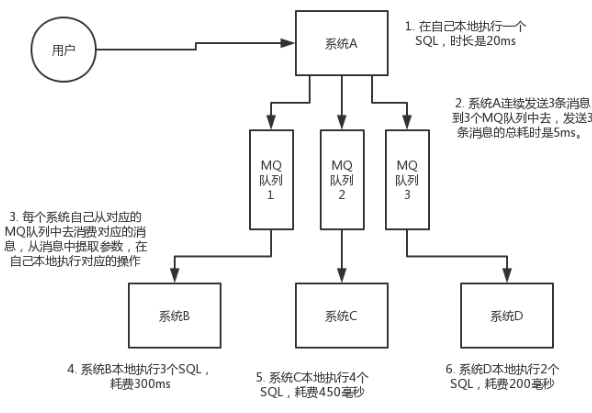
将消息写入消息队列，需要消息的系统自己从消息队列中订阅，从而系统A不需要做任何修改。

### 异步：

A系统接收一个请求，需要在自己本地写库，还需要在BCD三个系统写库，自己本地写库要3ms，BCD三个系统分别写库要300ms、450ms、200ms。最终请求总延时是3 + 300 + 450 + 200 = 953ms，接近1s，用户感觉搞个什么东西，慢死了慢死了。

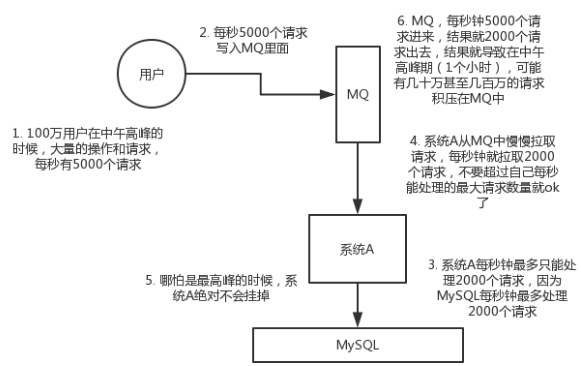
传统模式：

加入消息队列以后的中间件模式：



### 削峰：

每天0点到11点，A系统风平浪静，每秒并发请求数量就100个。结果每次一到11点~1点，每秒并发请求数量突然会暴增到1万条。但是系统最大的处理能力就只能是每秒钟处理1000个请求啊。。。



## 缺点

可以从两个角度来思考：

1.系统可用性降低：你想呀，本来其他系统只要运行好好的，那你的系统就是正常的。现在你非要加入个消息队列进去，那消息队列挂了，你的系统不是呵呵了。因此，系统可用性会降低

2.系统复杂性增加：加入了消息队列，要多考虑很多方面的问题，比如：一致性问题、如何保证消息不被重复消费、如何保证消息可靠性传输等。因此，需要考虑的东西更多，刺痛复杂性增大。

# 如何保证消息不被重复消费？

这个问题其实换一种问法就是，如何保证消息队列的幂等性？

### 造成重复消费的原因是什么？

正常情况下，消费者在消费消息的时候，消费完毕后，会发送一个确认消息给消息队列，消息队列就知道该消息被消费了，就会将该消息从消息队列中删除。但是因为网络传输等等故障，确认信息没有传送到消息队列，导致消息队列不知道自己已经消费过该消息了，再次将消息分发给其他的消费者。

如何解决？这个问题针对业务场景来答，分以下三种情况：

（1）比如，你拿到这个消息做数据库的insert操作，那就容易了，给这个消息做一个唯一的主键，那么就算出现重复消费的情况，就会导致主键冲突，避免数据库出现脏数据。

（2）再比如，你拿到这个消息做redis的set的操作，那就容易了，不用解决，因为你无论set几次结果都是一样的，set操作本来就算幂等操作。

（3）如果上面两种情况还不行，上大招。准备一个第三方介质，来做消费记录。以redis为例，给消息分配一个全局id，只要消费过该消息，将<id,message>以K-V形式写入redis。那消费者开始消费前，先去redis中查询有没有消费记录即可。

## 另外一种回答方式：

1. 消费端处理消息的业务逻辑保持幂等性。
2. 保证每条消息都有唯一编号且保证消息处理成功与去重表的日志同时出现。

第1条很好理解，只要保持幂等性，不管来多少条重复消息，最后处理的结果都一样。第2条原理就是利用一张日志表来记录已经处理成功的消息的ID，如果新到的消息ID已经在日志表中，那么就不再处理这条消息。

我们可以看到第1条的解决方式，很明显应该在消费端实现，不属于消息系统要实现的功能。第2条可以消息系统实现，也可以业务端实现。正常情况下出现重复消息的概率不一定大，且由消息系统实现的话，肯定会对消息系统的吞吐量和高可用有影响，所以最好还是由业务端自己处理消息重复的问题，这也是MQ不解决消息重复的问题的原因。

MQ不保证消息不重复，如果你的业务需要保证严格**的不重复消息，需要你自己在业务端去重。**

# 如何保证消息的可靠性传输？

消息不能重复消费，但也不能少消费。

其实这个可靠性传输，每种MQ都要从三个角度来分析：

* 生产者弄丢数据
* 消息队列弄丢数据
* 消费者弄丢数据

（1）生产者丢数据  
从生产者弄丢数据这个角度来看，RabbitMQ提供transaction和confirm模式来确保生产者不丢消息。

transaction模式：发送消息前，开启事务（channel.txSelect()）,然后发送消息，如果发送过程中出现什么异常，事务就会回滚（channel.txRollback()）,如果发送成功则提交事务（channel.txCommit()）。

开启事务-->发送消息--->异常--->回滚

--->成功---->提交

缺点：吞吐量下降。

confirm模式：按照经验，生产上用confirm模式的居多。一旦channel进入confirm模式，所有在该信道上发布的消息都将会被指派一个唯一的ID（从1开始），一旦**消息被投递到所有匹配的队列之后**，rabbitMQ就会发送一个ACK给生产者（包含消息的唯一ID），这就使得生产者知道消息已经正确到达目的队列了。如果rabbitMQ没能处理该消息，则会发送一个Nack消息给你，你可以进行重试操作。

（2）消息队列丢数据

处理消息队列丢数据的情况，一般是开启持久化磁盘的配置。这个持久化配置可以和confirm机制配合使用，你可以在消息持久化磁盘后，再给生产者发送一个Ack信号。这样，如果消息持久化磁盘之前，rabbitMQ阵亡了，那么生产者收不到Ack信号，生产者会自动重发。

如何持久化：

1. 将queue的持久化标识durable设置为true,则代表是一个持久的队列；
2. 发送消息的时候将deliveryMode=2。

这样设置以后，即使rabbitMQ挂了，重启后也能恢复数据。

（3）消费者丢数据

消费者丢数据一般是因为采用了自动确认消息模式。这种模式下，消费者会自动确认收到信息。这时rabbitMQ会立即将消息删除，这种情况下，如果消费者出现异常而未能处理消息，就会丢失该消息。

至于解决方案，采用手动确认消息即可。

# rabbitMQ的消息确认机制

首先它的确认机制分为两部分，一部分是生产端，一部分是消费端。有两种选择，一种是事务机制，一种是Confirm机制。

## 生产端的消息确认：

### RabbitMQ事务机制

RabbitMQ中与事务机制有关的方法有三个：txSelect(), txCommit()以及txRollback(), txSelect用于将当前channel设置成transaction模式，txCommit用于提交事务，txRollback用于回滚事务，在通过txSelect开启事务之后，我们便可以发布消息给broker代理服务器了，如果txCommit提交成功了，则消息一定到达了broker了，如果在txCommit执行之前broker异常崩溃或者由于其他原因抛出异常，这个时候我们便可以捕获异常通过txRollback回滚事务了。

事务确实能够解决producer与broker之间消息确认的问题，只有消息成功被broker接受，事务提交才能成功，否则我们便可以在捕获异常进行事务回滚操作同时进行消息重发，但是使用事务机制的话会降低RabbitMQ的性能。因此，RabbitMQ提供了一个更好的方案，即将channel信道设置成confirm模式。

### confirm模式

生产者将信道设置成confirm模式，一旦信道进入confirm模式，所有在该信道上面发布的消息都会被指派一个唯一的ID(从1开始)，一旦消息被投递到所有匹配的队列之后，broker就会发送一个确认给生产者（包含消息的唯一ID）,这就使得生产者知道消息已经正确到达目的队列了，如果消息和队列是可持久化的，那么确认消息会将消息写入磁盘之后发出，broker回传给生产者的确认消息中deliver-tag域包含了确认消息的序列号，此外broker也可以设置basic.ack的multiple域，表示到这个序列号之前的所有消息都已经得到了处理。

confirm模式最大的好处在于他是异步的，一旦发布一条消息，生产者应用程序就可以在等信道返回确认的同时继续发送下一条消息，当消息最终得到确认之后，生产者应用便可以通过回调方法来处理该确认消息，如果RabbitMQ因为自身内部错误导致消息丢失，就会发送一条nack消息，生产者应用程序同样可以在回调方法中处理该nack消息。

在channel 被设置成 confirm 模式之后，所有被 publish 的后续消息都将被 confirm（即 ack） 或者被nack一次。但是没有对消息被 confirm 的快慢做任何保证，并且同一条消息不会既被 confirm又被nack 。

事务机制和publisher confirm机制两者是互斥的，不能共存。

事务机制和publisher confirm机制确保的是消息能够正确的发送至RabbitMQ，这里的“发送至RabbitMQ”的含义是指消息被正确的发往至RabbitMQ的交换器，如果此交换器没有匹配的队列的话，那么消息也将会丢失。所以在使用这两种机制的时候要确保所涉及的交换器能够有匹配的队列。

## 消费端的消息确认：

为了保证消息从队列可靠地到达消费者，RabbitMQ提供消息确认机制(message acknowledgment)。消费者在声明队列时，可以指定noAck参数，当noAck=false时，RabbitMQ会等待消费者显式发回ack信号后才从内存(和磁盘，如果是持久化消息的话)中移去消息。否则，RabbitMQ会在队列中消息被消费后立即删除它。

采用消息确认机制后，只要令noAck=false，消费者就有足够的时间处理消息(任务)，不用担心处理消息过程中消费者进程挂掉后消息丢失的问题，因为RabbitMQ会一直持有消息直到消费者显式调用basicAck为止。RabbitMQ不会为未ack的消息设置超时时间，它判断此消息是否需要重新投递给消费者的唯一依据是消费该消息的消费者连接是否已经断开。这么设计的原因是RabbitMQ允许消费者消费一条消息的时间可以很久很久。

当noAck=false时，对于RabbitMQ服务器端而言，队列中的消息分成了两部分：一部分是等待投递给消费者的消息；一部分是已经投递给消费者，但是还没有收到消费者ack信号的消息。如果服务器端一直没有收到消费者的ack信号，并且消费此消息的消费者已经断开连接，则服务器端会安排该消息重新进入队列，等待投递给下一个消费者（也可能还是原来的那个消费者）。

# 如何保证消息的顺序性？

针对这个问题，通过某种算法，将需要**保持先后顺序的消息**放到**同一个消息队列中**( rabbitMq中就是queue)。然后只用一个消费者去消费该队列。  
那如果为了吞吐量，有多个消费者去消费怎么办？

这个问题，没有固定回答的套路。比如我们有一个微博的操作，发微博、写评论、删除微博，这三个异步操作。如果是这样一个业务场景，**那只要重试就行**。比如你一个消费者先执行了写评论的操作，但是这时候，微博都还没发，写评论一定是失败的，等一段时间。等另一个消费者，先执行写评论的操作后，再执行，就可以成功。

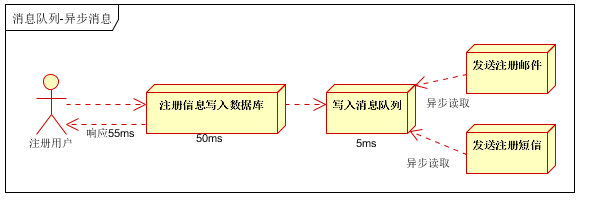
总之，针对这个问题，我的观点是保证**入队有序就行**，**出队**以后的顺**序交给消费者自己去保证**，没有固定套路。

# 消息队列的应用场景

### 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.并行方式。

引入消息队列，将不是必须的业务逻辑，异步处理。改造后的架构如下：



按照以上约定，用户的响应时间相当于是注册信息写入数据库的时间，也就是50毫秒。注册邮件，发送短信写入消息队列后，直接返回，因此写入消息队列的速度很快，基本可以忽略，因此用户的响应时间可能是50毫秒。

### 2应用解耦

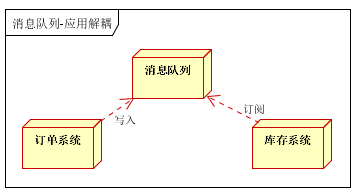
场景说明：用户下单后，订单系统需要通知库存系统。传统的做法是，订单系统调用库存系统的接口。

传统模式的缺点：

1）  假如库存系统无法访问，则订单减库存将失败，从而导致订单失败；

2）  订单系统与库存系统耦合；

引入应用消息队列后的方案，如下图：



 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，采用拉/推的方式，获取下单信息，库存系统根据下单信息，进行库存操作。

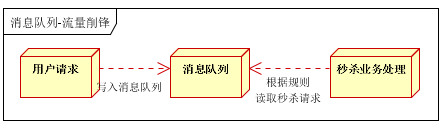
假如：在下单时库存系统不能正常使用。也不影响正常下单，因为下单后，订单系统写入消息队列就不再关心其他的后续操作了。实现订单系统与库存系统的应用解耦。

### 3.流量削锋

流量削锋也是消息队列中的常用场景，一般在秒杀或团抢活动中使用广泛。

应用场景：秒杀活动，一般会因为流量过大，导致流量暴增，应用挂掉。为解决这个问题，一般需要在应用前端加入消息队列。

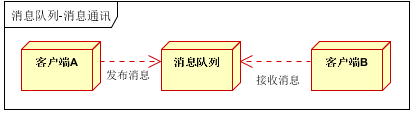
1. 可以控制活动的人数；
2. 可以缓解短时间内高流量压垮应用；



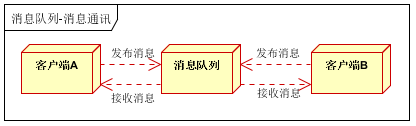
### 4.消息通讯

消息通讯是指，消息队列一般都内置了高效的通信机制，因此也可以用在纯的消息通讯。比如实现点对点消息队列，或者聊天室等。

点对点通讯：客户端A和客户端B使用同一队列，进行消息通讯。



聊天室通讯：客户端A，客户端B，客户端N订阅同一主题，进行消息发布和接收。实现类似聊天室效果。



以上实际是消息队列的两种消息模式，点对点或发布订阅模式。

# 其他小结

1.RabbitMQ 最优秀的功能之一就是内建集群，这个功能设计的目的是允许消费者和生产者在节点崩溃的情况下继续运行，以及通过添加更多的节点来线性扩展消息通信吞吐量。RabbitMQ 内部利用 Erlang 提供的分布式通信框架 OTP 来满足上述需求，使客户端在失去一个 RabbitMQ 节点连接的情况下，还是能够重新连接到集群中的任何其他节点继续生产、消费消息。

2.RabbitMQ 会始终记录以下四种类型的内部元数据：

* 队列元数据  
  包括队列名称和它们的属性，比如是否可持久化，是否自动删除
* 交换器元数据  
  交换器名称、类型、属性
* 绑定元数据  
  内部是一张表格记录如何将消息路由到队列
* vhost 元数据  
  为 vhost 内部的队列、交换器、绑定提供命名空间和安全属性