什么是消息中间件?

消息中间件(MQ)的定义

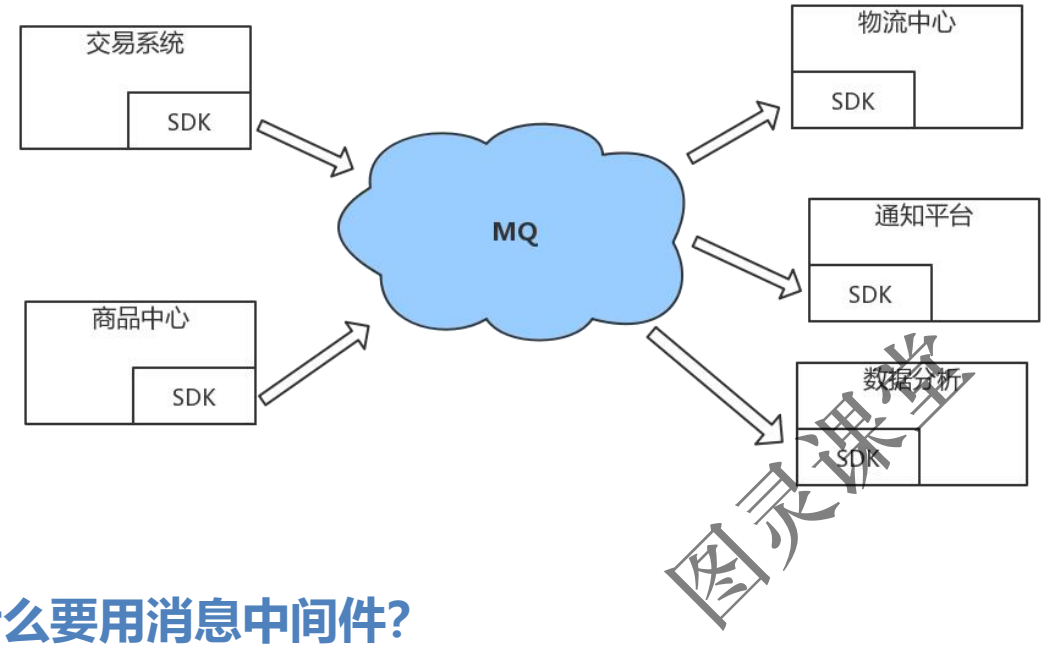
其实并没有标准定义。一般认为，消息中间件属于分布式系统中一个子系统，关注于数据的发送和接收，利用高效可靠的异步消息传递机制对分布式系统中的其余各个子系统进行集成。

高效：对于消息的处理处理速度快。

可靠：一般消息中间件都会有消息持久化机制和其他的机制确保消息不丢失。

异步：指发送完一个请求，不需要等待返回，随时可以再发送下一个请求，既不需要等待。

一句话总结，我们消息中间件不生产消息，只是消息的搬运工。



为什么要用消息中间件?

假设一个电商交易的场景，用户下单之后调用库存系统减库存，然后需要调用物流系统进行发货，如果交易、库存、物流是属于一个系统的，那么就是接口调用。但是随着系统的发展，各个模块越来越庞大、业务逻辑越来越复杂，必然是要做服务化和业务拆分的。这个时候就需要考虑这些系统之间如何交互，一般的处理方式就是 RPC（Remote Procedure Call）（具体实现 dubbo,SpringCloud）。系统继续发展，可能一笔交易后续需要调用几十个接口来执行业务，比如还有风控系统、短信服务等等。这个时候就需要消息中间件登场来解决问题了。

所以消息中间件主要解决分布式系统之间消息的传递，同时为分布式系统中其他子系统提供了松耦合的架构，同时还有以下好处：

低耦合

低耦合，不管是程序还是模块之间，使用消息中间件进行间接通信。

异步通信能力

异步通信能力，使得子系统之间得以充分执行自己的逻辑而无需等待。

缓冲能力

缓冲能力，消息中间件像是一个巨大的蓄水池，将高峰期大量的请求存储下来慢慢交给后台进行处理，对于秒杀业务来说尤为重要。

伸缩性

伸缩性，*是指通过不断向集群中加入服务器的手段来缓解不断上升的用户并发访问压力和不断增长的数据存储需求。就像弹簧一样挂东西一样，用户多，伸一点，用户少，浅一点，啊，不对，缩一点。是伸缩，不是深浅。衡量架构是否高伸缩性的主要标准就是是否可用多台服务器构建集群，是否容易向集群中添加新的服务器。加入新的服务器后是否可以提供和原来服务器无差别的服务。集群中可容纳的总的服务器数量是否有限制。*

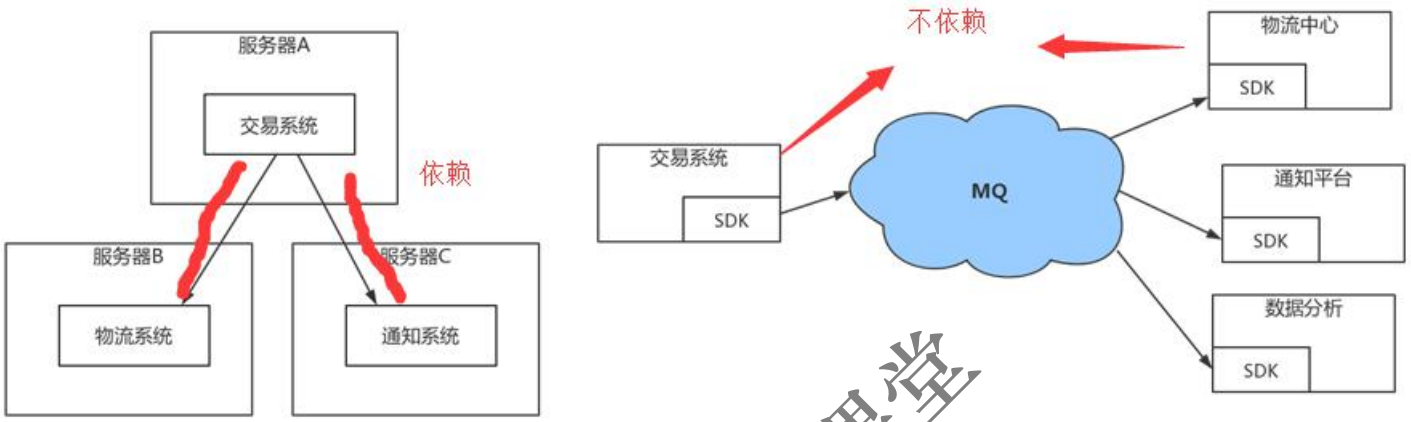
扩展性

扩展性，*主要标准就是在网站增加新的业务产品时，是否可以实现对现有产品透明无影响，不需要任何改动或者很少改动既有业务功能就可以上线新产品。比如用户购买电影票的应用，现在我们要增加一个功能，用户买了铁血战士的票后，随机抽取用户送异形的限量周边。怎么做到不改动用户购票功能的基础上增加这个功能。熟悉设计模式的同学，应该很眼熟，这是设计模式中的开闭原则（对扩展开放，对修改关闭）在架构层面的一个原则。*

和 RPC 有何区别?

RPC 和消息中间件的场景的差异很大程度上在于就是“依赖性”和“同步性”。

依赖性：

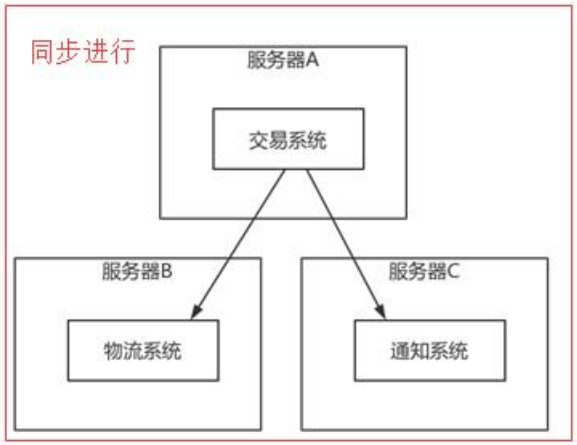


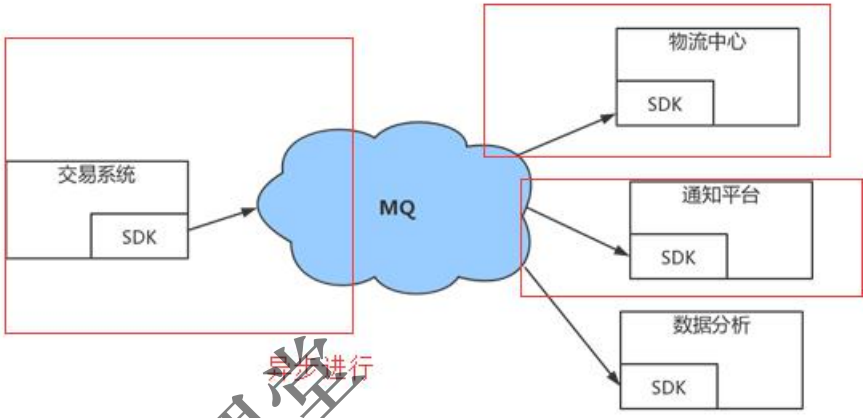
比如短信通知服务并不是事交易环节必须的，并不影响下单流程，不是强依赖，所以交易系统不应该依赖短信服务。如果是 RPC 调用，短信通知服务挂了，整个业务就挂了，这个就是依赖性导致的，而消息中间件则没有这个依赖性。

消息中间件出现以后对于交易场景可能是调用库存中心等强依赖系统执行业务，之后发布一条消息（这条消息存储于消息中间件中）。像是短信通知服务、数据统计服务等等都是依赖于消息中间件去消费这条消息来完成自己的业务逻辑。

同步性：

RPC 方式是典型的同步方式，让远程调用像本地调用。消息中间件方式属于异步方式。





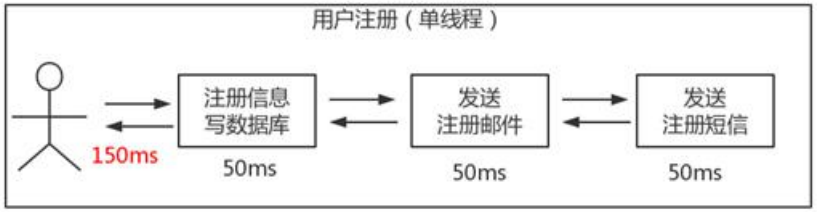
相同点：都是分布式下面的通信方式。

消息中间件有些什么使用场景?

异步处理

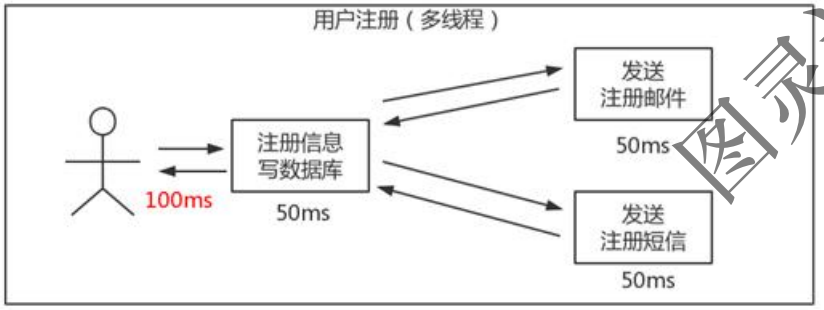
场景说明：用户注册后，需要发注册邮件和注册短信。传统的做法有两种 1.串行的方式；2.并行方式。

串行方式：将注册信息写入数据库成功后，发送注册邮件，再发送注册短信。以上三个任务全部完成后，返回给客户端。



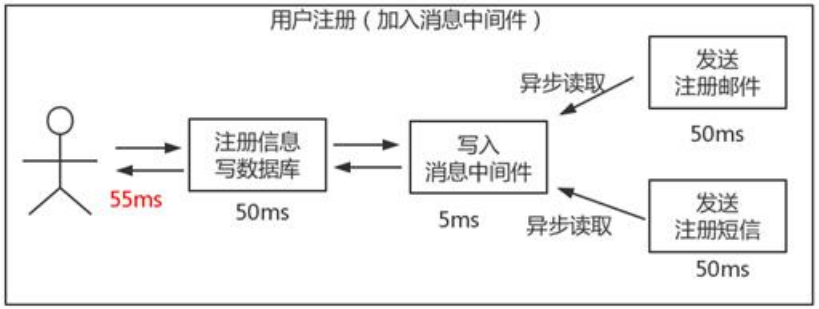
（2）并行方式：将注册信息写入数据库成功后，发送注册邮件的同时，发送注册短信。以上三个任务完成后，返回给客户端。与串行的差别是，并行的方式可以提高处理的时间。

假设三个业务节点每个使用 50 毫秒钟，不考虑网络等其他开销，则串行方式的时间是 150 毫秒，并行的时间可能是 100 毫秒。



小结：如以上案例描述，传统的方式系统的性能（并发量，吞吐量，响应时间）会有瓶颈。如何解决这个问题呢？

引入消息队列，将不是必须的业务逻辑，异步处理。



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。因此架构改变后，系统的吞吐量提高到每秒 20 QPS。比串行提高了 3 倍，比并行提高了两倍。

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：

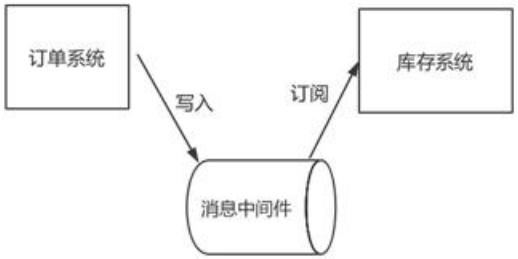
1） 假如库存系统无法访问，则订单减库存将失败，从而导致订单失败；

2） 订单系统与库存系统耦合；

如何解决以上问题呢？引入应用消息队列后的方案

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。

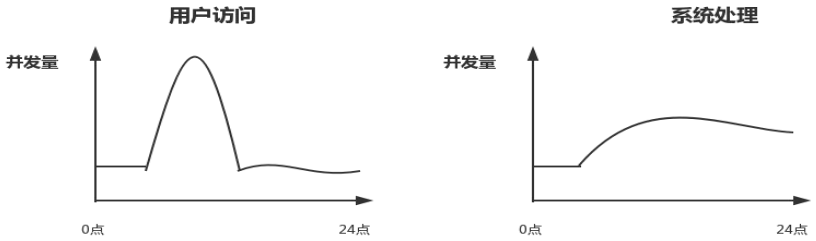


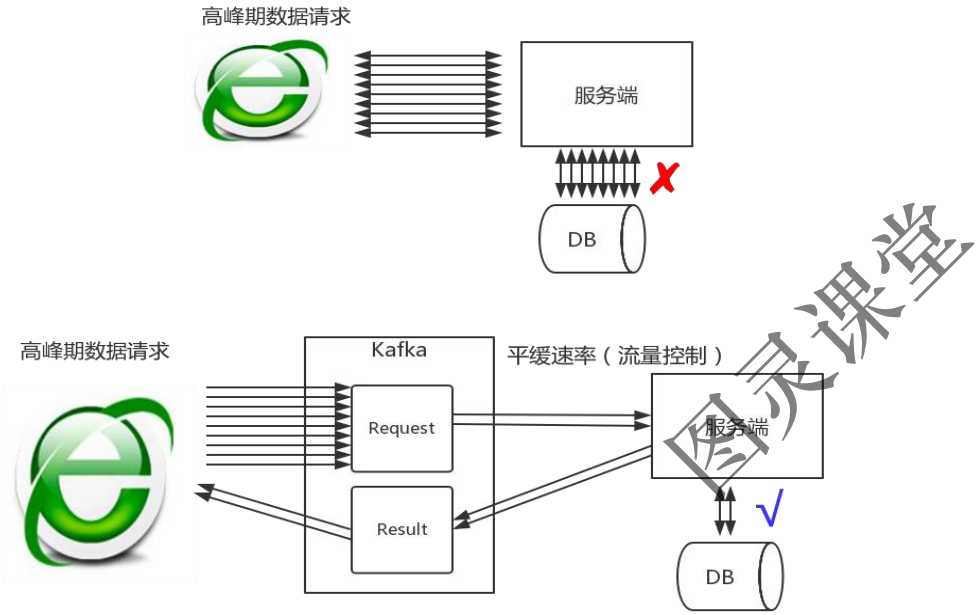
假如：在下单时库存系统不能正常使用。也不影响正常下单，因为下单后，订单系统写入消息队列就不再关心其他的后续操作了。实现订单系统与库存系统的应用解耦。

流量削峰

流量削峰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：可以控制活动的人数；可以缓解短时间内高流量压垮应用。





用户的请求，服务器接收后，首先写入消息队列。假如消息队列长度超过最大数量，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。

日志处理

日志处理是指将消息队列用在日志处理中，比如 Kafka 的应用，解决大量日志传输的问题。架构简化如下：

日志采集客户端，负责日志数据采集，定时写入 Kafka 队列：Kafka 消息队列，负责日志数据的接收，存储和转发；日志处理应用：订阅并消费 kafka 队列中的日志数据；

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。

消息中间件的编年史

卡夫卡与法国作家马塞尔·普鲁斯特，爱尔兰作家詹姆斯·乔伊斯并称为西方现代主义文学的先驱和大师。《变形记》是卡夫卡的短篇代表作，是卡夫卡的艺术成就中的一座高峰，被认为是 20 世纪最伟大的小说作品之一。

常见的消息中间件比较

	ActiveMQ	RabbitMQ	RocketMQ	Kafka
性能（单台）	6000+	万级(12000+)	十万级	百万级
消息持久化	支持	支持	支持	支持
多语言支持	支持	支持	很少	支持
社区活跃度	高	高	中	高
支持协议	多（JMS,AMQP..... ）	多（AMQP,STOMP,MQTT..... ）	自定义协议	自定义协议
综合评价	优点： 成熟，已经在很多公司得到应用。较多的文档。各种协议支持较好，有多个语言的成熟客户端。 缺点： 性能较弱。缺乏大规模吞吐的场景的应用，有江河日下之感。	优点：性能较好，管理界面较丰富，在互联网公司也有较大规模的应用，有多个语言的成熟客户端。 缺点： 内部机制很难理解，也意味很难定制和掌控。集群不支持动态扩展。	优点：模型简单，接口易用。在阿里有大规模应用。分布式系统，性能很好，版本更新很快。 缺点：文档少，支持的语言较少。	优点：天生分布式，性能最好，所以常见用于大数据领域。 缺点：运维难度大，偶尔有数据混乱的情况，对ZooKeeeper强依赖。多副本机制下对带宽有一定的要求。

如果一般的业务系统要引入 MQ，怎么选型：

用户访问量在 ActiveMQ 的可承受范围内，而且确实主要是基于解耦和异步来用的，可以考虑 ActiveMQ，也比较贴近 Java 工程师的使用习惯，但是 ActiveMQ 现在停止维护了，同时 ActiveMQ 并发不高，所以业务量一定的情况下可以考虑使用。

RabbitMQ 作为一个纯正血统的消息中间件，有着高级消息协议 AMQP 的完美结合，在消息中间件中地位无可取代，但是 erlang 语言阻止了我们去深入研究和掌控，对公司而言，底层技术无法控制，但是确实是开源的，有比较稳定的支持，活跃度也高。

对自己公司技术实力有绝对自信的，可以用 RocketMQ 。

所以中小型公司，技术实力较为一般，技术挑战不是特别高，用 ActiveMQ、RabbitMQ 是不错的选择；大型公司，基础架构研发实力较强，用 RocketMQ 是很好的选择

如果是大数据领域的实时计算、日志采集等场景，用 Kafka 是业内标准的，绝对没问题，社区活跃度很高，几乎是全世界这个领域的事实性规范。

整体上来看，使用文件系统的消息中间件（kafka、rokcetMq）性能是最好的，所以基于文件系统存储的消息中间件是发展趋势。（从存储方式和效率来看 文件系统>KV 存储>关系型数据库）

AMQP 概论

AMQP

是应用层协议的一个开放标准,为面向消息的中间件设计。基于此协议的客户端与消息中间件可传递消息，并不受客户端/中间件不同产品，不同的开发语言等条件的限制。目标是实现一种在全行业广泛使用的标准消息中间件技术，以便降低企业和系统集成的开销，并且向大众提供工业级的集成服务。主要实现有 RabbitMQ。

客户端与 RabbitMQ 的通讯

连接

首先作为客户端（无论是生产者还是消费者），你如果要与 RabbitMQ 通讯的话，你们之间必须创建一条 TCP 连接，当然同时建立连接后，客户端还必须发送一条“问候语”让彼此知道我们都是符合 AMQP 的语言的，比如你跟别人打招呼一般会说“你好！”，你跟国外的美女一般会说“hello!”一样。你们确认好“语言”之后，就相当于客户端和 RabbitMQ 通过“认证”了。你们之间可以创建一条 AMQP 的信道。

信道

概念：信道是生产者/消费者与 RabbitMQ 通信的渠道。信道是建立在 TCP 连接上的虚拟连接，什么意思呢？就是说 rabbitmq 在一条 TCP 上建立成百上千个信道来达到多个线程处理，这个 TCP 被多个线程共享，每个线程对应一个信道，信道在 RabbitMQ 都有唯一的 ID ,保证了信道私有性，对应上唯一的线程使用。

疑问：为什么不建立多个 TCP 连接呢？原因是 rabbit 保证性能，系统为每个线程开辟一个 TCP 是非常消耗性能，每秒成百上千的建立销毁 TCP 会严重消耗系统。所以 rabbitmq 选择建立多个信道（建立在 tcp 的虚拟连接）连接到 rabbit 上。

从技术上讲，这被称之为“多路复用”，对于执行多个任务的多线程或者异步应用程序来说，它非常有用。

RabbitMQ 中使用 AMQP

包括的要素

生产者、消费者、消息

生产者:消息的创建者，发送到 rabbitmq；

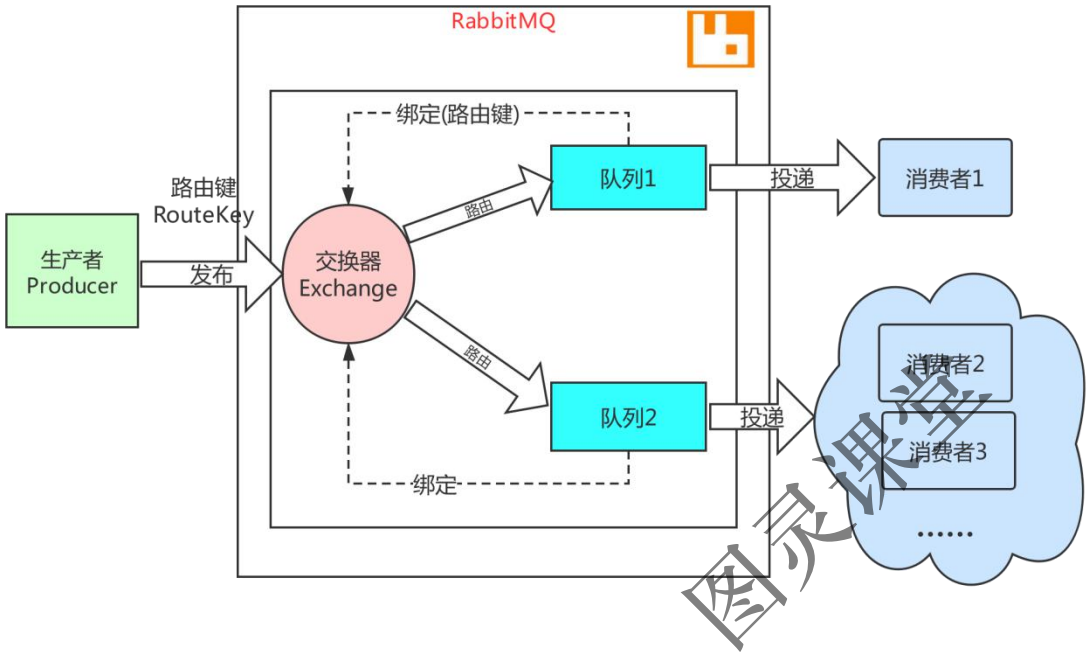
消费者：连接到 rabbitmq，订阅到队列上，消费消息，持续订阅(basicConsumer)和单条订阅(basicGet).

消息：包含有效载荷和标签，有效载荷指要传输的数据，，标签描述了有效载荷，并且 rabbitmq 用它来决定谁获得消息，消费者只能拿到有效载荷，并不知道生产者是谁。

交换器、队列、绑定、路由键

队列通过路由键（routing　key，某种确定的规则）绑定到交换器，生产者将消息发布到交换器，交换器根据绑定的路由键将消息路由到特定队列，然后由订阅这个队列的消费者进行接收。

（routing_key 和 绑定键 binding_key 的最大长度是 255 个字节）



消息的确认

消费者收到的每一条消息都必须进行确认（自动确认和自行确认）。

消费者在声明队列时，可以指定 autoAck 参数，当 autoAck=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 autoAck=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。

当 autoAck=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）。

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。

常见问题

如果消息达到无人订阅的队列会怎么办？消息会一直在队列中等待，RabbitMq 默认队列是无限长度的。

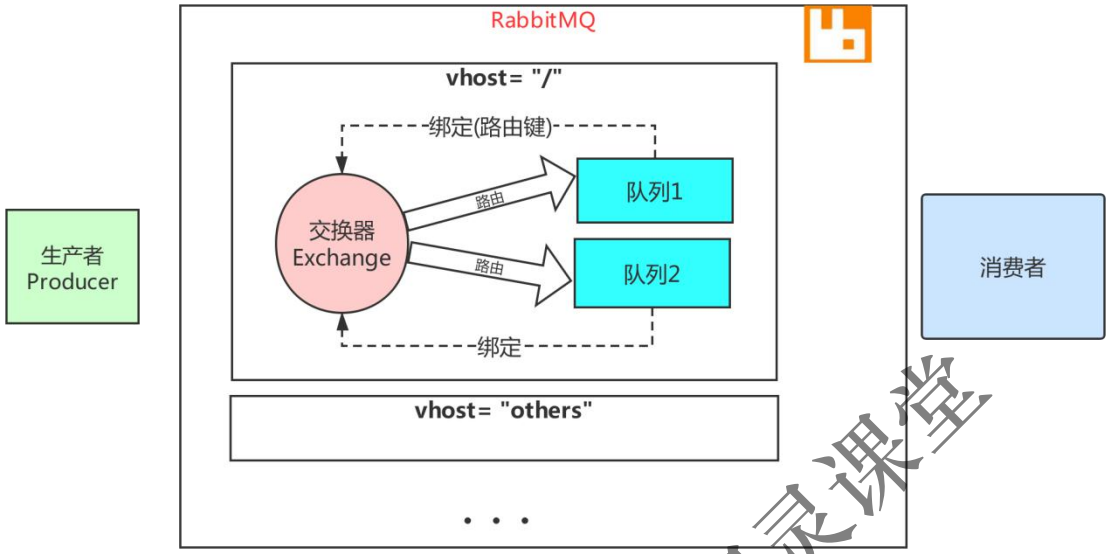
多个消费者订阅到同一队列怎么办？消息以循环的方式发送给消费者，每个消息只会发送给一个消费者。

消息路由到了不存在的队列怎么办？一般情况下，凉拌，RabbitMq 会忽略，当这个消息不存在，也就是这消息丢了。

虚拟主机

虚拟消息服务器，vhost，本质上就是一个 mini 版的 mq 服务器，有自己的队列、交换器和绑定，最重要的，自己的权限机制。Vhost 提供了逻辑上的分离，可以将众多客户端进行区分，又可以避免队列和交换器的命名冲突。Vhost 必须在连接时指定，rabbitmq 包含缺省 vhost：“/”，通过缺省用户和口令 guest 进行访问。

rabbitmq 里创建用户，必须要被指派给至少一个 vhost，并且只能访问被指派内的队列、交换器和绑定。Vhost 必须通过 rabbitmq 的管理控制工具创建。

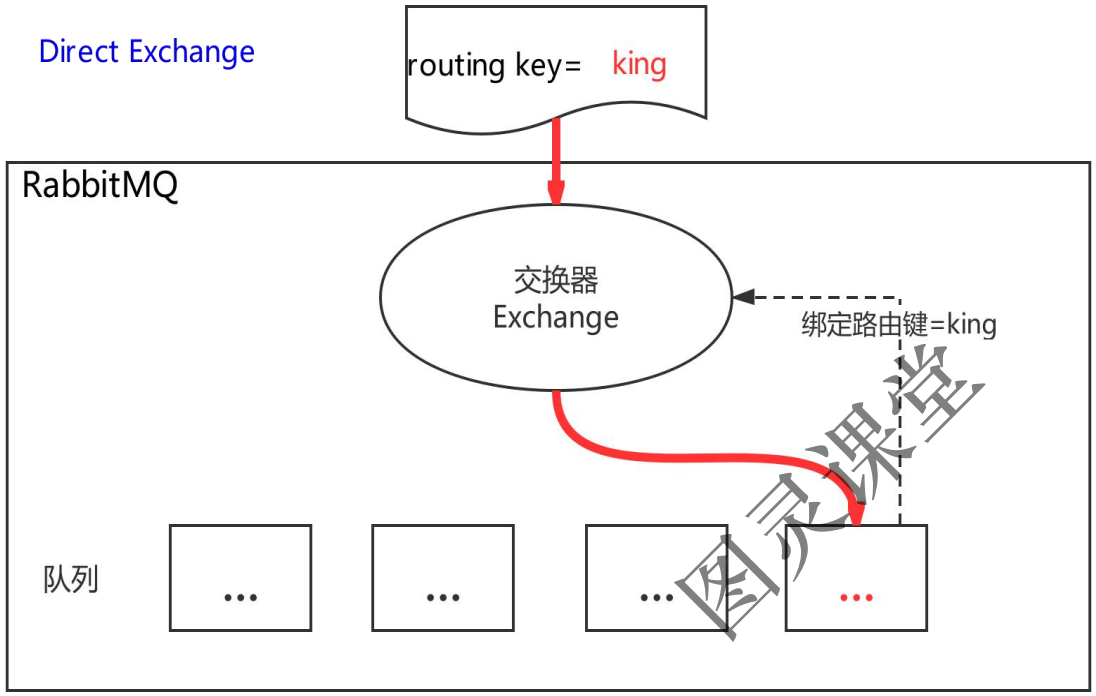


交换器类型

共有四种 direct,fanout,topic,headers，其种 headers(几乎和 direct 一样)不实用，可以忽略。

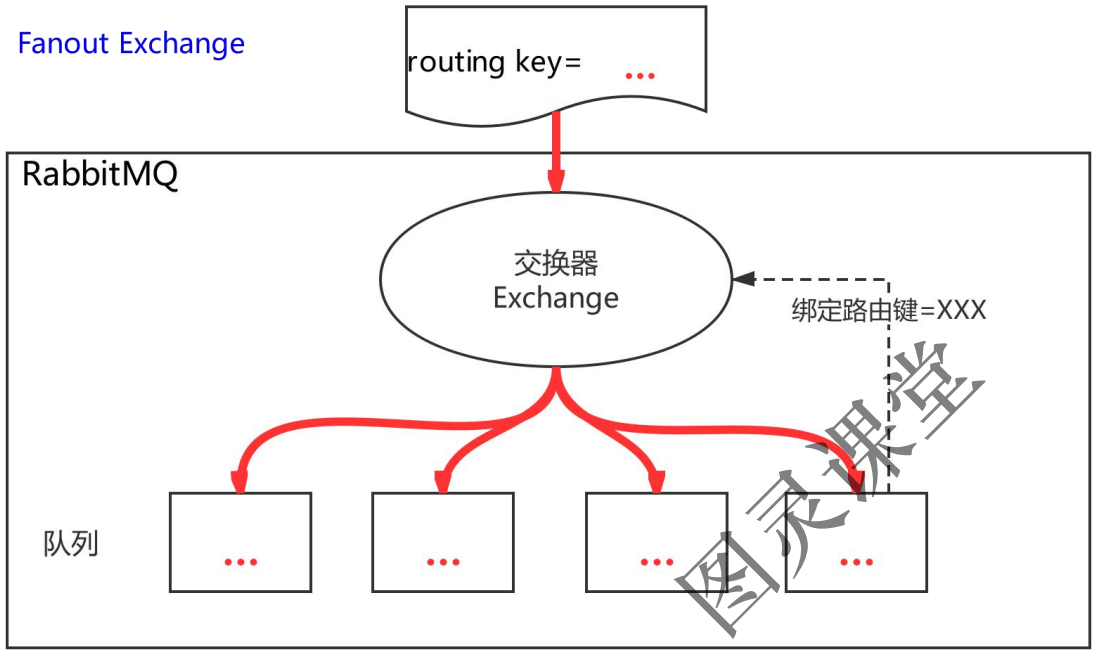
Direct

路由键完全匹配，消息被投递到对应的队列， direct 交换器是默认交换器。声明一个队列时，会自动绑定到默认交换器，并且以队列名称作为路由键：channel->basic_public($msg,’’,’queue-name’)



Fanout

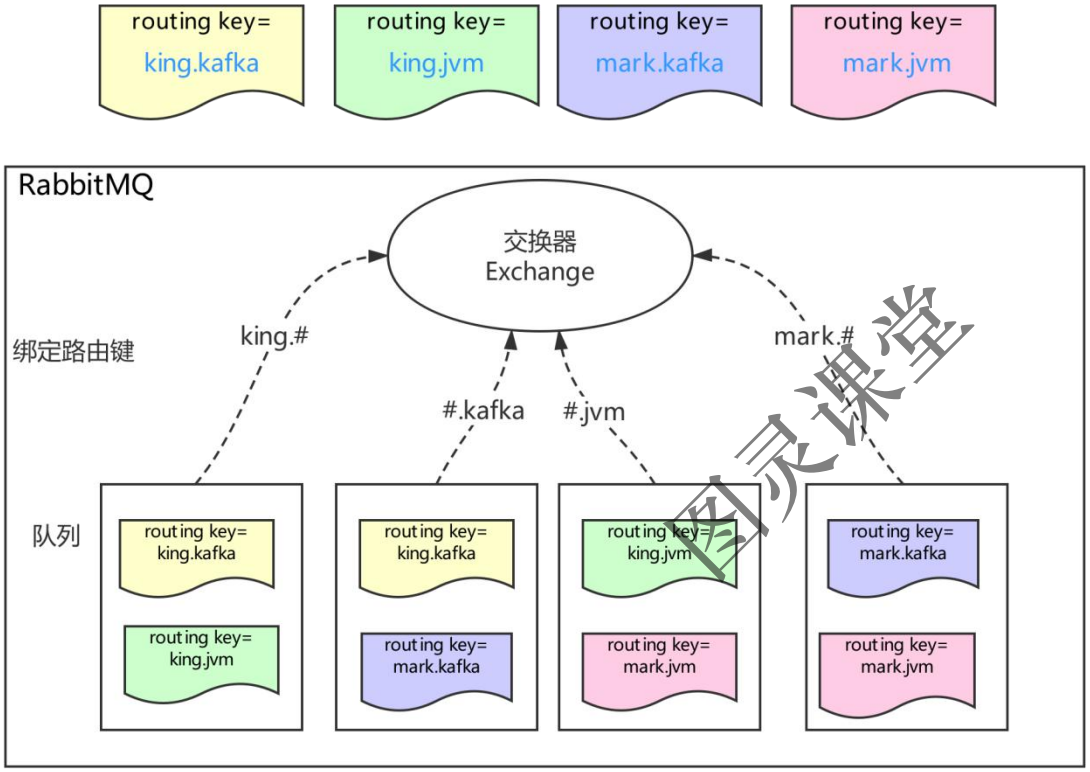
消息广播到绑定的队列，不管队列绑定了什么路由键，消息经过交换器，每个队列都有一份。



Topic

通过使用“*”和“#”通配符进行处理，使来自不同源头的消息到达同一个队列，”.”将路由键分为了几个标识符，“*”匹配 1 个，“#”匹配一个或多个。

Topic Exchange



RabbitMQ 在 Windows 下安装和运行

见《RabbitMQ 的安装.docx》

原生 Java 客户端进行消息通信

Direct 交换器

参见代码 native 模块包 cn.enjoyedu.exchange.direct 中：

DirectProducer：***direct** 类型交换器的生产者*

NormalConsumer：*普通的消费者*

MulitBindConsumer：*队列绑定到交换器上时，是允许绑定多个路由键的，也就是多重绑定*

MulitChannelConsumer：*一个连接下允许有多个信道*

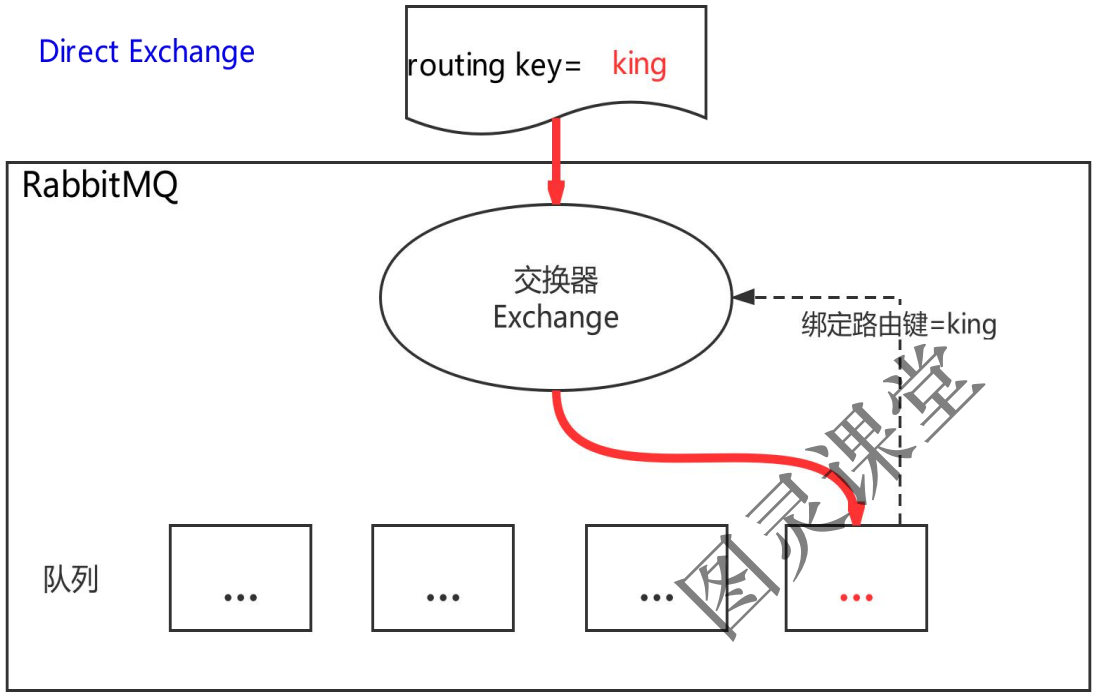
MulitConsumerOneQueue：*一个队列多个消费者，则会表现出消息在消费者之间的轮询发送。*

生产者和消费者一般用法

DirectProducer：***direct** 类型交换器的生产者*

NormalConsumer：*普通的消费者*

使用 DirectProducer 作为生产者，NormalConsumer 作为消费者，消费者绑定一个队列



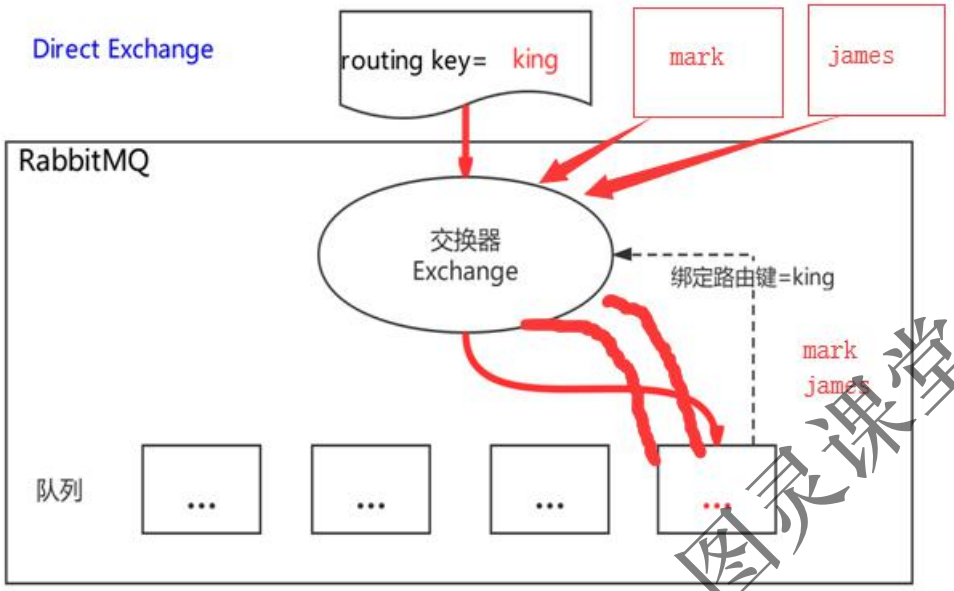
队列和交换器的多重绑定

DirectProducer：***direct** 类型交换器的生产者*

NormalConsumer：*普通的消费者*

MultiBindConsumer：*队列绑定到交换器上时，是允许绑定多个路由键的，也就是多重绑定*

对比：单个绑定的消费者只能收到指定的消息，多重绑定的的消费者可以收到所有的消息。



一个连接多个信道

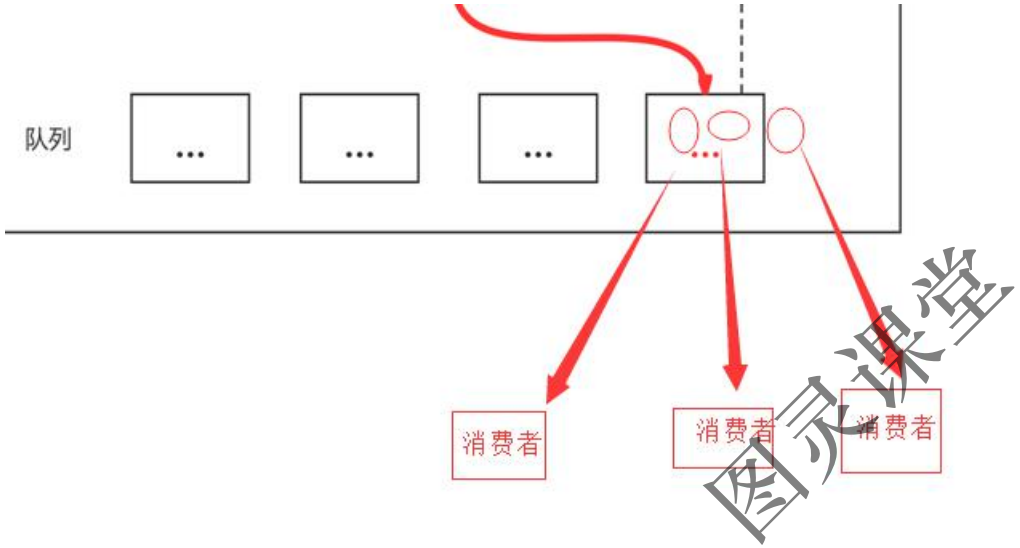
DirectProducer：***direct** 类型交换器的生产者*

MultiChannelConsumer：*一个连接下允许有多个信道*

一个连接，我们可以使用多线程的方式模拟多个信道进行通讯。这样可以做到多路复用。

一个队列多个消费者

MultiConsumerOneQueue： *一个队列多个消费者，则会表现出消息在消费者之间的轮询发送。*



Fanout

消息广播到绑定的队列

参见代码 native 模块包 cn.enjoyedu.exchange.fanout 中：

通过测试表明，不管我们如何调整生产者和消费者的路由键，都对消息的接受没有影响。

Topic

参见代码 native 模块包 cn.enjoyedu.exchange.topic 中：

通过使用“*”和“#”，使来自不同源头的消息到达同一个队列，”.”将路由键分为了几个标识符，“*”匹配 1 个，“#”匹配一个或多个。例子如下：

假设有交换器 topic_course，

讲课老师有 king,mark,james，

技术专题有 kafka,jvm,redis，

课程章节有 A、B、C，

路由键的规则为 讲课老师+“.”+技术专题+“.”+课程章节，如：king.kafka.A。

1、要关注所有的课程，怎么做?

声明一个队列并绑定到交换器上：channel.queueBind(queueName,TopicProducer.EXCHANGE_NAME, "#");

2、关注 king 老师的所有课程，怎么办?

声明一个队列并绑定到交换器上：channel.queueBind(queueName,TopicProducer.EXCHANGE_NAME, "king.#");

注意:如果这里改为 king.* 的话，则不会出现任何信息，因为“*”匹配 1 个（使用.分割的标识的个数）

3、关注 king 老师所有的 A 章节，怎么办?

声明一个队列并绑定到交换器上：channel.queueBind(queueName,TopicProducer.EXCHANGE_NAME, "king.#.A");

或者声明一个队列并绑定到交换器上：channel.queueBind(queueName,TopicProducer.EXCHANGE_NAME, "king.*.A");

4、关注 kafka 所有的课程，怎么办?

声明一个队列并绑定到交换器上：channel.queueBind(queueName,TopicProducer.EXCHANGE_NAME, "#.kafka.#");

5、关注所有的B章节，怎么办?

声明一个队列并绑定到交换器上：channel.queueBind(queueName,TopicProducer.EXCHANGE_NAME, "#.B");

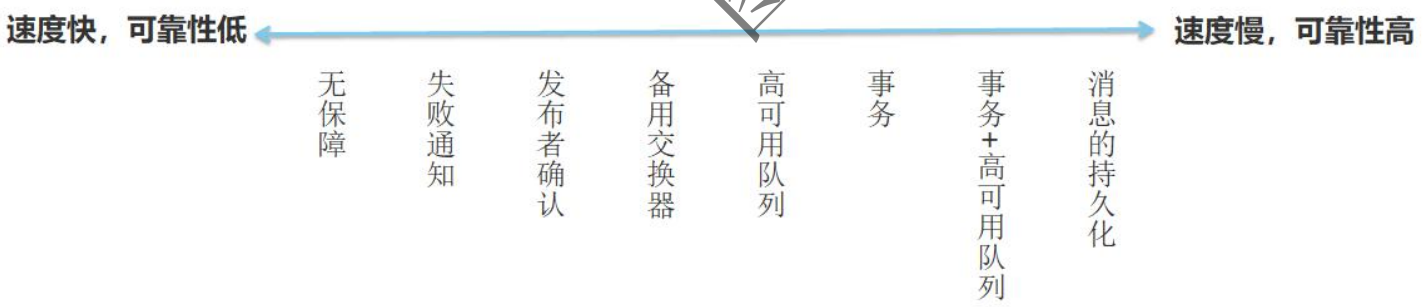
6、关注king老师kafka的A章节，怎么办?

声明一个队列并绑定到交换器上：channel.queueBind(queueName,TopicProducer.EXCHANGE_NAME, "king.kafka.A");

消息发布时的权衡

在 RabbitMQ 在设计的时候，特意让生产者和消费者“脱钩”，也就是消息的发布和消息的消费之间是解耦的。

在 RabbitMQ 中，有不同的投递机制（生产者），但是每一种机制都对性能有一定的影响。一般来讲速度快的可靠性低，可靠性好的性能差，具体怎么使用需要根据你的应用程序来定，所以说没有最好的方式，只有最合适的方式。只有把你的项目和技术相结合，才能找到适合你的平衡。



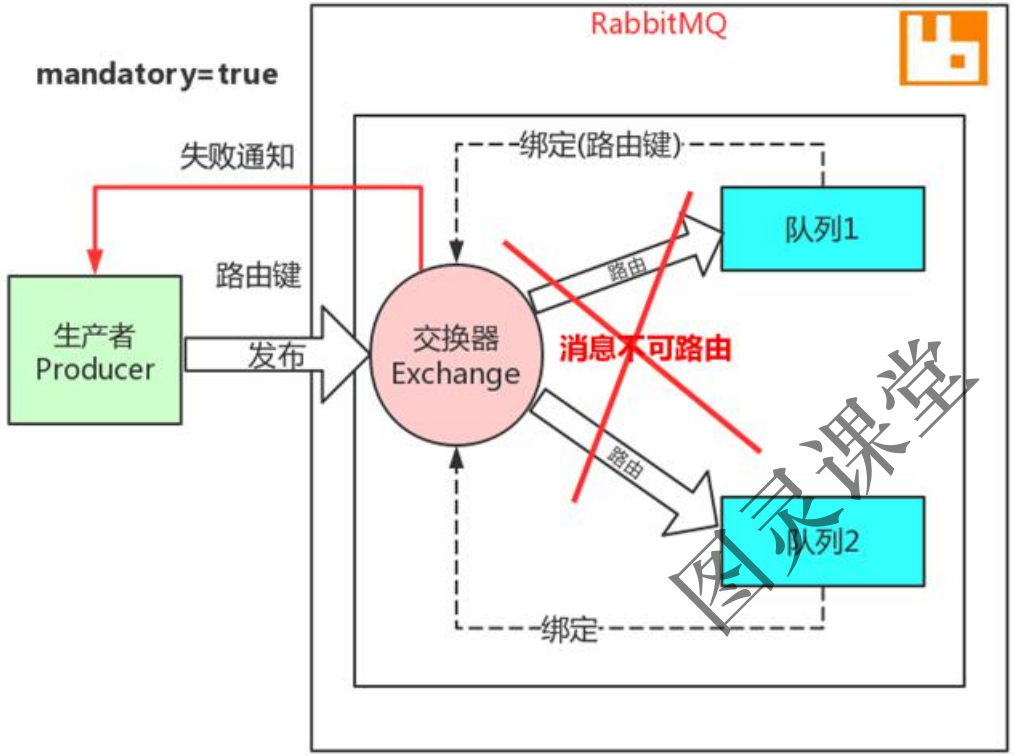
无保障

在演示各种交换器中使用的就是无保障的方式，通过 basicPublish 发布你的消息并使用正确的交换器和路由信息，你的消息会被接收并发送到合适的队列中。但是如果有网络问题，或者消息不可路由，或者 RabbitMQ 自身有问题的话，这种方式就有风险。所以无保证的消息发送一般情况下不推荐。

失败确认

在发送消息时设置 mandatory 标志，告诉 RabbitMQ，如果消息不可路由，应该将消息返回给发送者，并通知失败。可以这样认为，开启 mandatory 是开启故障检测模式。

注意：它只会让 RabbitMQ 向你通知失败，而不会通知成功。如果消息正确路由到队列，则发布者不会受到任何通知。带来的问题是无法确保发布消息一定是成功的，因为通知失败的消息可能会丢失。



channel.addConfirmListener 则用来监听 RabbitMQ 发回的信息。

如何使用，参见代码 native 模块包 cn.enjoyedu.producer_balance.mandatory 中。

监听器的小甜点

在信道关闭和连接关闭时，还有两个监听器可以使用

```
connection.addShutdownListener(new ShutdownListener() {
    public void shutdownCompleted(ShutdownSignalException cause) {
        System.out.println("connect is shutdown: "+cause.getMessage()
                +System.currentTimeMillis());
    }
});

channel.addShutdownListener(new ShutdownListener() {
    public void shutdownCompleted(ShutdownSignalException cause) {
        System.out.println("channel is shutdown: "+cause.getMessage()
                +System.currentTimeMillis());
    }
});
```

事务

事务的实现主要是对信道（Channel）的设置，主要的方法有三个：

1. channel.txSelect()声明启动事务模式；
2. channel.txComment()提交事务；
3. channel.txRollback()回滚事务；

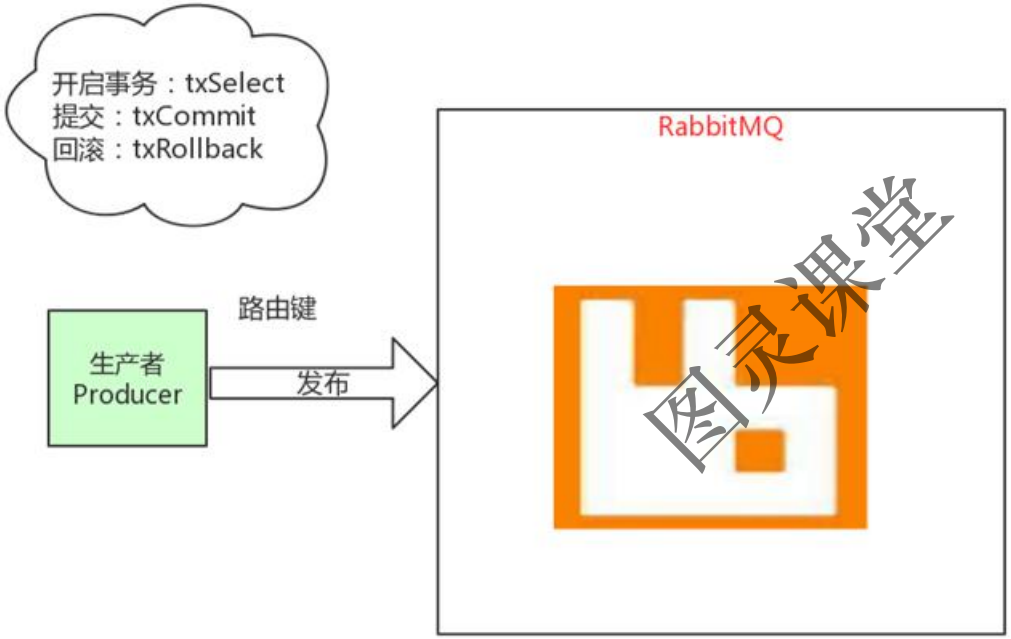
在发送消息之前，需要声明 channel 为事务模式，提交或者回滚事务即可。

开启事务后，客户端和 RabbitMQ 之间的通讯交互流程：

- 客户端发送给服务器 Tx.Select(开启事务模式)
- 服务器端返回 Tx.Select-Ok（开启事务模式 ok）
- 推送消息
- 客户端发送给事务提交 Tx.Commit

- 服务器端返回 Tx.Commit-Ok

以上就完成了事务的交互流程，如果其中任意一个环节出现问题，就会抛出 IoException 移除，这样用户就可以拦截异常进行事务回滚，或决定要不要重复消息。



那么，既然已经有事务了，为何还要使用发送方确认模式呢，原因是因为事务的性能是非常差的。根据相关资料，事务会降低 2~10 倍的性能。

如何使用，参见代码 native 模块包 cn.enjoyedu.producer_balance.transaction 中。

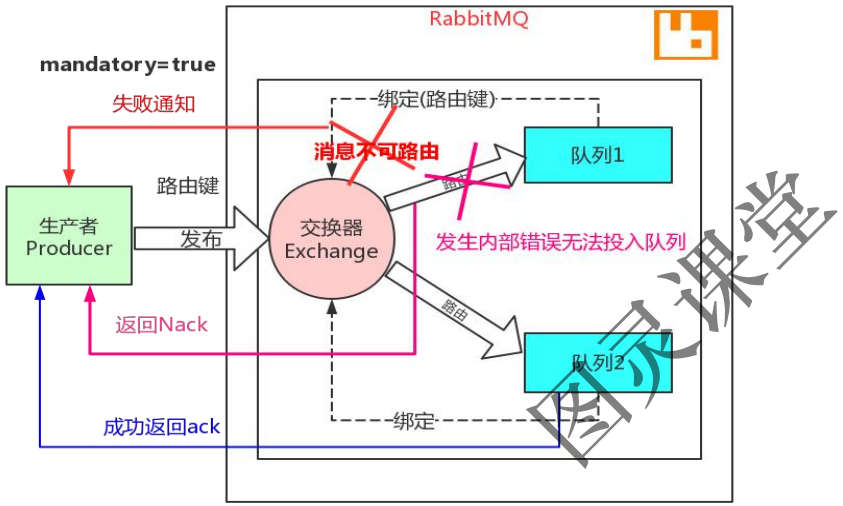
发送方确认模式

基于事务的性能问题，RabbitMQ 团队为我们拿出了更好的方案，即采用**发送方确认模式，**该模式比事务更轻量，性能影响几乎可以忽略不计。

原理：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将会被指派一个唯一的 ID(从 1 开始)，由这个 id 在生产者和 RabbitMQ 之间进行消息的确认。

不可路由的消息，当交换器发现，消息不能路由到任何队列，会进行确认操作，表示收到了消息。如果发送方设置了 mandatory 模式,则会先调用 addReturnListener 监听器。

可路由的消息，要等到消息被投递到所有匹配的队列之后，broker 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，那么确认消息会在将消息写入磁盘之后发出，broker 回传给生产者的确认消息中 delivery-tag 域包含了确认消息的序列号。



confirm 模式最大的好处在于他可以是异步的，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 消息决定下一步的处理。

Confirm 的三种实现方式：

方式一：channel.waitForConfirms()普通发送方确认模式；消息到达交换器，就会返回 true。

方式二：channel.waitForConfirmsOrDie()批量确认模式；使用同步方式等所有的消息发送之后才会执行后面代码，只要有一个消息未到达交换器就会抛出 IOException 异常。

方式三：channel.addConfirmListener()异步监听发送方确认模式；

如何使用，参见代码 native 模块包 cn.enjoyedu.producer_balance.producerconfirm 中。

备用交换器

在第一次声明交换器时被指定，用来提供一种预先存在的交换器，如果主交换器无法路由消息，那么消息将被路由到这个新的备用交换器。

如果发布消息时同时设置了 mandatory 会发生什么？如果主交换器无法路由消息，RabbitMQ 并不会通知发布者，因为，向备用交换器发送消息，表示消息已经被路由了。注意，新的备用交换器就是普通的交换器，没有任何特殊的地方。

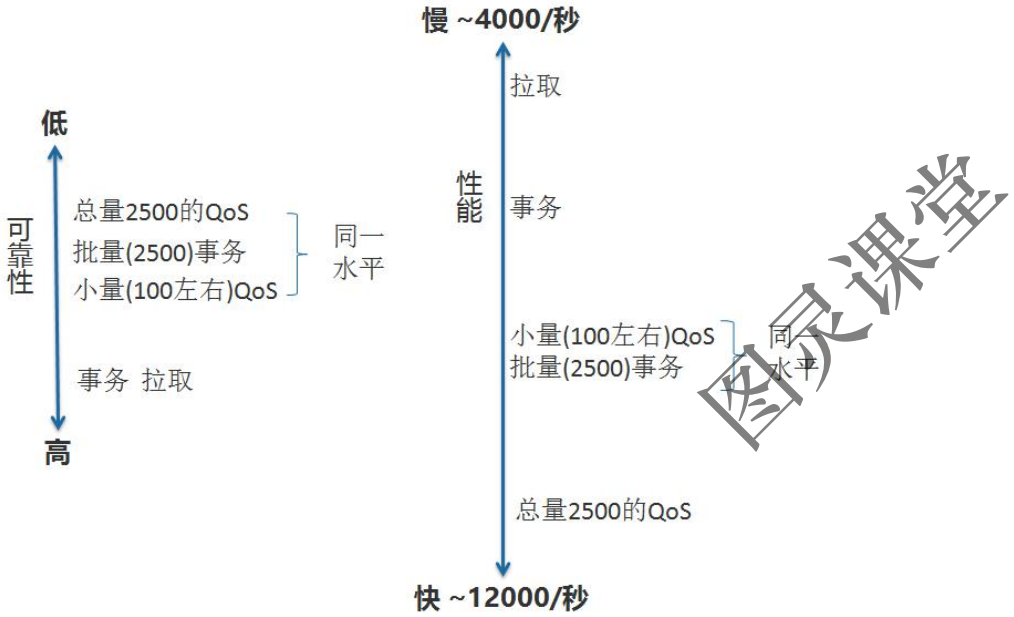
使用备用交换器，向往常一样，声明 Queue 和备用交换器，把 Queue 绑定到备用交换器上。然后在声明主交换器时，通过交换器的参数，alternate-exchange,，将备用交换器设置给主交换器。

建议备用交换器设置为 faout 类型，Queue 绑定时的路由键设置为“#”

如何使用，参见代码 native 模块包 cn.enjoyedu.producer_balance.backupexchange 中。

消息的消费

可靠性和性能的权衡



消息的获得方式

拉取 Get

属于一种轮询模型，发送一次 get 请求，获得一个消息。如果此时 RabbitMQ 中没有消息，会获得一个表示空的回复。总的来说，这种方式性能比较差，很明显，每获得一条消息，都要和 RabbitMQ 进行网络通信发出请求。而且对 RabbitMQ 来说，RabbitMQ 无法进行任何优化，因为它永远不知道应用程序何时会发出请求。具体使用，参见代码 native 模块包 cn.enjoyedu.consumer_balance.GetMessage 中。对我们实现者来说，要在一个循环里，不断去服务器 get 消息。

推送 Consume

属于一种推送模型。注册一个消费者后，RabbitMQ 会在消息可用时，自动将消息进行推送给消费者。这种模式我们已经使用过很多次了，具体使用，参见代码 native 模块包 cn.enjoyedu.exchange.direct 中。

消息的应答

前面说过，消费者收到的每一条消息都必须进行确认。消息确认后，RabbitMQ 才会从队列删除这条消息，RabbitMQ 不会为未确认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。

自动确认

消费者在声明队列时，可以指定 autoAck 参数，当 autoAck=true 时，一旦消费者接收到了消息，就视为自动确认了消息。如果消费者在处理消息的过程中，出了错，就没有什么办法重新处理这条消息，所以我们很多时候，需要在消息处理成功后，再确认消息，这就需要手动确认。

手动确认

当 autoAck=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 autoAck=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。

当 autoAck=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）。

如何使用，参见代码 native 模块包 cn.enjoyedu.consumer_balance.ackfalse 中。

通过运行程序，启动两个消费者 A、B，都可以收到消息，但是其中有一个消费者 A 不会对消息进行确认，当把这个消费者 A 关闭后，消费者 B 又会收到本来发送给消费者 A 的消息。所以我们一般使用手动确认的方法是，将消息的处理放在 try/catch 语句块中，成功处理了，就给 RabbitMQ 一个确认应答，如果处理异常了，就在 catch 中，进行消息的拒绝，如何拒绝，参考消息的拒绝章节。

QoS 预取模式

在确认消息被接收之前，消费者可以预先要求接收一定数量的消息，在处理完一定数量的消息后，批量进行确认。如果消费者应用程序在确认消息之前崩溃，则所有未确认的消息将被重新发送给其他消费者。所以这里存在着一定程度上的可靠性风险。

这种机制一方面可以实现限速（将消息暂存到 RabbitMQ 内存中）的作用，一方面可以保证消息确认质量（比如确认了但是处理有异常的情况）。

注意：消费确认模式必须是非自动 ACK 机制（这个是使用 baseQos 的前提条件，否则会 Qos 不生效），然后设置 basicQos 的值；另外，还可以基于 consume 和 channel 的粒度进行设置（global）。

具体使用，参见代码 native 模块包 cn.enjoyedu.consumer_balance.qos 中。我们可以进行批量确认，也可以进行单条确认。

basicQos 方法参数详细解释：

prefetchSize：最多传输的内容的大小的限制，0 为不限制，但据说 prefetchSize 参数，rabbitmq 没有实现。

prefetchCount：会告诉 RabbitMQ 不要同时给一个消费者推送多于 N 个消息，即一旦有 N 个消息还没有 ack，则该 consumer 将 block 掉，直到有消息 ack

global：true\false 是否将上面设置应用于 channel，简单点说，就是上面限制是 channel 级别的还是 consumer 级别。

如果同时设置 channel 和消费者，会怎么样？AMQP 规范没有解释如果使用不同的全局值多次调用 basic.qos 会发生什么。 RabbitMQ 将此解释为意味着两个预取限制应该彼此独立地强制执行；消费者只有在未达到未确认消息限制时才会收到新消息。

```
channel.basicQos(10, false); // Per consumer limit
channel.basicQos(15, true);  // Per channel limit
channel.basicConsume("my-queue1", false, consumer1);
channel.basicConsume("my-queue2", false, consumer2);
```

也就是说，整个通道加起来最多允许 15 条未确认的消息，每个消费者则最多有 10 条消息。

消费者中的事务

使用方法和生产者一致

假设消费者模式中使用了事务，并且在消息确认之后进行了事务回滚，会是什么样的结果？

结果分为两种情况：

1. autoAck=false 手动应对的时候是支持事务的，也就是说即使你已经手动确认了消息已经收到了，但 RabbitMQ 对消息的确认会等事务的返回结果，再做最终决定是确认消息还是重新放回队列，如果你手动确认之后，又回滚了事务，那么以事务回滚为准，此条消息会重新放回队列；
2. autoAck=true 如果自动确认为 true 的情况是不支持事务的，也就是说你即使在收到消息之后在回滚事务也是于事无补的，队列已经把消息移除了。

消息的拒绝

Reject 和 Nack

消息确认可以让 RabbitMQ 知道消费者已经接受并处理完消息。但是如果消息本身或者消息的处理过程出现问题怎么办？需要一种机制，通知 RabbitMQ，这个消息，我无法处理，请让别的消费者处理。这里就有两种机制，Reject 和 Nack。

Reject 在拒绝消息时，可以使用 requeue 标识，告诉 RabbitMQ 是否需要重新发送给别的消费者。不重新发送，一般这个消息就会被 RabbitMQ 丢弃。Reject 一次只能拒绝一条消息。

Nack 则可以一次性拒绝多个消息。这是 RabbitMQ 对 AMQP 规范的一个扩展。

具体使用，参见代码 native 模块包 cn.enjoyedu. rejectmsg 中。通过 RejectRequeuConsumer 可以看到当 requeue 参数设置为 true 时，消息发生了重新投递。

死信交换器 DLX

RabbitMQ 对 AMQP 规范的一个扩展。被投递消息被拒绝后的一个可选行为，往往用在对问题消息的诊断上。

消息变成死信一般是以下几种情况：

- 消息被拒绝，并且设置 requeue 参数为 false
- 消息过期
- 队列达到最大长度

死信交换器仍然只是一个普通的交换器，创建时并没有特别要求和操作。在创建队列的时候，声明该交换器将用作保存被拒绝的消息即可，相关的参数是 x-dead-letter-exchange。

具体使用，参见代码 native 模块包 cn.enjoyedu.dlx 中。

通过运行程序可以看到，生产者产生了 3 条消息，分别是 error,info,warn，两个消费者 WillMakeDlxConsumer 和 WillMakeWarnDlxConsumer 都拒绝了两条消息，而投送到死信队列后，可以发现根据投送死信时的路由键，不同的消费者有些可以接受到消息，有些则不行。

和备用交换器的区别

1、备用交换器是主交换器无法路由消息，那么消息将被路由到这个新的备用交换器，而死信交换器则是接收过期或者被拒绝的消息。

2、备用交换器是在声明主交换器时发生联系，而死信交换器则声明队列时发生联系。

延时队列的实现

控制队列

参见代码 native 模块包 cn.enjoyedu.setQueue 中

临时队列

自动删除队列

自动删除队列和普通队列在使用上没有什么区别，唯一的区别是，当消费者断开连接时，队列将会被删除。自动删除队列允许的消费者没有限制，也就是说当这个队列上最后一个消费者断开连接才会执行删除。

自动删除队列只需要在声明队列时，设置属性 auto-delete 标识为 true 即可。系统声明的随机队列，缺省就是自动删除的。

```
queueDeclare(String queue, boolean durable, boolean exclusive, boolean autoDelete,
             Map<String, Object> arguments) throws IOException;
```

设为true

单消费者队列

普通队列允许的消费者没有限制，多个消费者绑定到多个队列时，RabbitMQ 会采用轮询进行投递。如果需要消费者独占队列，在队列创建的时候，设定属性 exclusive 为 true。

```
queueDeclare(String queue, boolean durable, boolean exclusive, boolean autoDelete,
             Map<String, Object> arguments) throws IOException;
```

设为true

自动过期队列

指队列在超过一定时间没使用，会被从 RabbitMQ 中被删除。什么是没使用？

一定时间内没有 Get 操作发生

没有 Consumer 连接在队列上

特别的：就算一直有消息进入队列，也不算队列在被使用。

通过声明队列时，设定 x-expires 参数即可，单位毫秒。


```
/*声明一个队列*/
String queueName = "setqueue";
Map<String, Object> arguments = new HashMap<~>();
/*队列在超过一定时间没使用，会被从RabbitMQ中被删除*/
arguments.put("x-expires",45*1000);
/*设定了该队列所有消息的存活时间，时间单位是毫秒*/
arguments.put("x-message-ttl ",40*1000);
channel.queueDeclare(queueName, durable: false, exclusive: false,
        autoDelete: false,arguments);
```

队列的参数map

参数赋给队列

永久队列

队列的持久性

持久化队列和非持久化队列的区别是，持久化队列会被保存在磁盘中，固定并持久的存储，当 Rabbit 服务重启后，该队列会保持原来的状态在 RabbitMQ 中被管理，而非持久化队列不会被保存在磁盘中，Rabbit 服务重启后队列就会消失。

非持久化比持久化的优势就是，由于非持久化不需要保存在磁盘中，所以使用速度就比持久化队列快。即是非持久化的性能要高于持久化。而持久化的优点就是会一直存在，不会随服务的重启或服务器的宕机而消失。

在声明队列时，将属性 durable 设置为“false”，则该队列为非持久化队列，设置成“true”时，该队列就为持久化队列

```
queueDeclare(String queue, boolean durable, boolean exclusive, boolean autoDelete,
             Map<String, Object> arguments) throws IOException;
```

设为true

队列级别消息过期

就是为每个队列设置消息的超时时间。只要给队列设置 x-message-ttl 参数，就设定了该队列所有消息的存活时间，时间单位是毫秒。如果声明队列时指定了死信交换器，则过期消息会成为死信消息。

```java
/*声明一个队列*/
String queueName = "setqueue";
Map<String, Object> arguments = new HashMap<~>();
/*队列在超过一定时间没使用，会被从RabbitMQ中被删除*/
arguments.put("x-expires",45*1000);
/*设定了该队列所有消息的存活时间，时间单位是毫秒*/
arguments.put("x-message-ttl ",40*1000);
channel.queueDeclare(queueName, durable: false, exclusive: false,
        autoDelete: false,arguments);
```

队列保留参数列表

参数名	目的
x-dead-letter-exchange	死信交换器
x-dead-letter-routing-key	死信消息的可选路由键
x-expires	队列在指定毫秒数后被删除
x-ha-policy	创建 HA 队列
x-ha-nodes	HA 队列的分布节点
x-max-length	队列的最大消息数

x-message-ttl	毫秒为单位的消息过期时间，队列级别
x-max-priority	最大优先值为 255 的队列优先排序功能