# 通信模型

## 同步通信模型





## 异步通信模型

http://www.zuidaima.com/blog/4398239732780032.htm

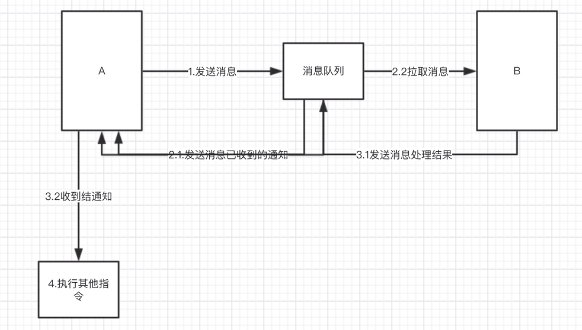
### 线程池



缺点：自己实现线程池，并且强耦合。

### 消息队列

消息队列使得同步通信变成异步通信：



这样就避免了前面发送消息和处理消息的耗时，这里只需要发送消息到消息队列，消息队列存储消息（不处理消息），然后迅速返回给客户端，然后才由接收端拉取消息处理。

#### 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，如下：



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，从而导致下单失败；
2. 订单系统与库存系统耦合。

引入消息队列后的方案，如下图：



订阅系统：用户下单后，订阅系统完成持久化处理，将消息写入消息队列，返回用户，下单成功。

库存系统：订阅下单的消息，获取下单信息后，库存系统根据下单信息，进行库存操作。

假如再下单时库存系统不能正常工作，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**秒杀或团购**活动中广泛应用。

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，作用如下：

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。



用户的请求，服务器接收后，首先写入消息队列，加入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消息，再做后续处理。

#### 异步处理



场景说明：用户注册后，需要发送注册邮件和注册短信，传统的做法如下：

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。



引入消息队列，处理异步，改造后架构：



### 联系/区别

1、两者内部都使用了队列，如阻塞队列、优先级队列；

2、使用线程池时应用服务器既充当生产者又充当消费者，也是消息队列中间件的实现者，使用消息队列时中间件、生产者、消费者可以部署在不同的应用机器上（当然也可以部署在一台服务器上但很少有人这么用）；

3、出于第2点线程池更适合非分布式的系统，但在分布式架构下消息队列明显是更优项；

4、使用消息队列会带来额外的网络开销；

5、消息队列的耦合性更低，可扩展性更好，适用于弱一致性的场景，如对log日志的解耦；

6、消息队列自动实现消息的持久化，中间已经实现了大量功能，如消息转发、消息拒绝、消息重试，以及对消息的一些监控，例如消息的消费状态、消息的消费速率、消息内容查询等，使用线程池如果需要很多功能还要自己去实现，例如想要知道执行状态还需要打印队列数量、计算消息消费速度；

7、在不同系统间的服务调用（调用协议也可能不一致）线程池很难实现或开销很大，这时候消息队列可以屏蔽不同机器或不同协议的问题；

8、使用消息队列会提升系统的复杂度，网络抖动怎么办？最大队列长度怎么设置？超时时间又设置多少？Qos又设置为多少？消费者多少个比较合适？Channel cache size又该设置为多少？业务线可能都是用同一个Mq，你占资源太多，或者设计不当可能会导致整个Mq故障。

# JMS消息模型

JMS=Java Message Service

## P2P对点对模式

P2P（Point to Point）模式包含三个角色：消息队列（Queue），发送者（Sender），接收者（Reciver）。

每个消息都被发送到一个特定的队列，接收者从队列中获取消息。队列保留着消息，直到它们被消费或者超时。

P2P的特点：

1. 每个消息只有一个消费者（即一旦被消费，消息就不再在消息队列中）；
2. 发送者和接收者之间在时间上没有依赖性，也就是说当发送者发送了消息之后，不管接收者有没有正在运行，它不会影响到消息被发送到队列；
3. 接收者在成功接收到消息之后需向队列应答成功。

**如果希望发送的每个消息都会被成功处理的话，那么需要P2P模式**。

## Publish/Subscribe发布订阅模式

Pub/Sub模式包含三个角色：主体（Topic），发布者（Publisher），订阅者（Subscriber）。多个发布者将消息发送到Topic，系统将这些消息传递给多个订阅者。

Pub/Sub的特点：

1. 每个消息可以有多个消费者；
2. **发布者和订阅者之间有时间上的依赖性**，针对某个主体（topic）的订阅者，它必须创建一个订阅之后，才能消费发布者的消息；
3. 为了消费消息，订阅者必须保持运行的状态。

为了缓和这样严格的时间相关性，JMS允许订阅者创建一个可持久化的订阅。这样，即使订阅者没有被激活（运行），它也能接收到发布者的消息。

如果希望发送的消息可以被多个消费者处理的话，那么可以采用Pub/Sub模型。

# 消息中间件

消息中间件可以解决如下问题：异步处理，应用解耦，流量削峰，日志处理，从而实现高性能，高可用，可伸缩和最终一致性的架构。

## 背景

生产者和消费者速度或稳定性等因素的不一致。

## 概述

MQ（Message Queue，消息队列）是一种应用程序对应用程序的通信方法。应用程序通过读写出入队列的消息（针对应用程序的数据）来通信，而无需专用链接来连接它们。

消息传递指的是程序之间通过在消息中发送数据进行通信，而不是通过直接调用彼此来通信，直接调用通常是用于远程过程调用技术。排队指的是应用程序通过队列来通信，队列的使用出去了接收和发送应用程序同时执行的要求。

### Provider

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。

没有使用消息队列时消息传递方式：



使用消息队列后消息传递方式：



### 队列

队列就像是存放了商品的仓库，是生产商品的工厂和购买商品的用户之间的中转站。

**队列里存储了什么？**

在RabbitMQ中，信息流从你的应用程序出发，来到RabbitMQ的队列，所有信息可以只存储在一个队列中。队列可以存储很多信息，因为它基本上是一个无限制的缓冲区，前提是你的机器有足够的存储空间。

**队列和应用程序的关系：**

多个生产者可以将消息发送到同一个队列中，多个消费者也可以只从同一个队列接收数据。

## 特性

**消息队列的特性：**

1. 业务无关：只做消息分发
2. FIFO：先投递先到达
3. 容灾：节点的动态增删和消息的持久化
4. 性能：吞吐量提升，系统内部通信效率提高

### 优点

1. 业务耦合；
2. 最终一致性（记录+补偿机制实现）；
3. 广播；
4. 错峰与流控。

### 缺点

1）系统更复杂，多了一个MQ组件

2）消息传递路径更长，延时会增加

3）消息可靠性和重复性互为矛盾，消息不丢不重难以同时保证

4）上游无法知道下游的执行结果，这一点是很致命的。

## 使用场景

<https://blog.csdn.net/weixin_41141219/article/details/80643461>

**什么时候不使用MQ？**

上游实时关注执行结果（比如数据库连接池，使用的是线程池而不是MQ）。

**什么时候使用MQ？**

1）数据驱动的任务依赖

2）上游不关心多下游执行结果

3）异步返回执行时间长

## ActiveMQ

### 概述

ActiveMQ是Apache出品，最流行的，能力强劲的开源消息总线，并且它一个完全支持JMS规范的消息中间件。

其丰富的API、多种集群构建模式使得他称为业界老牌消息中间件，在小型企业中应用广泛（**并发不是很大的场景**），但是对于高并发、大数据的场景就力不从心了。

MQ衡量指标：服务性能、数据存储、集群架构

### 原理

#### Topic消息失败重发

消息失败是指消息的消费方没有成功读取消息，此时需要重发消息，最终保持事务完整性和数据一致性。

JMS消息确认机制：

在session接口中定义的几个常量：

AUTO\_ACKNOWLEDGE = 1，自动确认

CLIENT\_ACKNOWLEDGE = 2，客户端手动确认

DUPS\_OK\_ACKNOWLEDGE = 3，自动批量确认

SESSION\_TRANSACTED = 0，事务提交并确认

代码实现：

消息消费端在创建session对象时需要指定应答模式位客户端手动应答，当消费者获取到消息并成功处理后需要调用message.achnowledge()方法进行应答，通知Broker消费成功。如果处理过程中出现异常，需要调用session.recover()通知Broker重复消息，默认最多重复6次。

#### Topic消息持久化订阅

##### 未持久化

必须先启动消费端，然后启动客户端发送消息到topic。如果客户端先向topic发送消息，然后启动消费端则无法消费该消息。此时发送端和消费端存在时间顺序。

##### 持久化到文件（默认）

基本步骤：

1. 在${activemq.base}/conf/activemq.xml文件中配置持久化适配器；
2. 在Java代码中发送消息时：
3. 消息订阅者方创建消费者对象时：

connection.setClientID()”client1-sub”;

TopicSubscriber consumer = session.createDurableSubscriber(topic,’client1-sub’);

##### 持久化到数据库

基本步骤：

1. 将MySQL的数据库驱动复制到ActiveMQ的lib目录下；
2. 在${activemq.base}/conf/activemq.xml文件中配置持久化适配器；
3. 在${activemq.base}/conf/activemq.xml文件中配置数据源。

### 架构

#### Master-Slave模式



#### Network模式



### Zookeeper+ActiveMQ

使用zookeeper实现的master-slave实现方式，是对ActiveMQ进行高可用的一种有效的解决方案。

高可用的原理：使用zookeeper（集群）注册所有的ActiveMQ Broker。只有其中的一个Broker可以对外提供服务（也就是Master节点），其他的Broker处于待机状态，被视为Slave。如果Master因故障而不能提供服务，则利用zookeeper的内部选举机制=从slave中选举出一个Broker充当Master节点，继续对外提供服务。通过zookeeper+activemq实现的集群，可以有效的排除单点故障引起的服务中断。

## Kafka

### 概述

Kafka是LinkedIn开源的分布式分布-订阅消息系统，目前归属于Apache顶级项目。

Kafka主要特点是基于Pull的模式来处理消息消费，追求高吞吐量，一开始的目的就是用于日志收集和传输（大数据场景）。0.8版本开始支持复制，不支持事务，对消息的重复、丢失、错误没有严格要求，适合产生大量数据的互联网服务的数据收集业务。

### 集群模式



## RocketMQ

### 概述

RocketMQ是阿里开源的消息中间件，目前已经孵化为Apache顶级项目，它是纯Java开发，具有高吞吐量、高可用性、适合大规模分布式系统应用的特点。

RocketMQ思路起源于Kafka，它对消息的可靠传输及事务性做了优化，目前在阿里集团被广泛应用于交易、充值、流计算、消息推送、日志流式处理、binglog分发等场景。

### 集群拓扑



## RabbitMQ

### 概述

RabbitMQ是使用Erlang语言开发的开源消息队列系统，基于AMQP协议来实现。AMQP的主要特征是面向消息、队列、路由（包括点对点和分布订阅）、可靠性、安全。AMQP协议更多用在企业系统内，对数据**一致性、稳定性和可靠性**要求很高的场景，对性能和吞吐量的要求还在其次。

### 集群



## 选型

中小型公司，技术实力较为一般，技术挑战不是特别高，用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。

Redis更适合用来做很小规模、业务简单的消息队列场景。 如果业务复杂、业务规模大，一般情况下Redis就会被排除。

ActiveMQ是Apache下的一个子项目。类似于ZeroMQ，它能够以代理人和点对点的技术实现队列。同时类似于RabbitMQ，它少量代码就可以高效地实现高级应用场景。

RabbitMQ是使用Erlang编写的一个开源的消息队列，本身支持很多的协议：AMQP，XMPP，SMTP，STOMP，也正因如此，它非常重量级，更适合于企业级的开发。同时实现了Broker构架，这意味着消息在发送给客户端时先在中心队列排队。对路由，负载均衡或者数据持久化都有很好的支持。

Kafka是Apache下的一个子项目，是一个高性能跨语言分布式发布/订阅消息队列系统。它具有以下特性：快速持久化，可以在O(1)的系统开销下进行消息持久化；高吞吐，在一台普通的服务器上既可以达到10W/s的吞吐速率；完全的分布式系统，Broker、Producer、Consumer都原生自动支持分布式，自动实现负载均衡；支持Hadoop数据并行加载，对于像 Hadoop的一样的日志数据和离线分析系统，但又要求实时处理的限制，这是一个可行的解决方案。

RocketMQ是阿里巴巴开源的一个项目，目前已经纳入Apache基金会。其是在Kafka 的基础上发展起来的，起因是随着阿里巴巴业务的发展，他们发现Kafka对于具体业务场景的支持不完善，所以才有了RocketMQ的诞生。

与Kafka比起来，RocketMQ很多方面都极其相似。唯一的不同是RocketMQ对于业务特性的支持更完善，所以更适用于业务场景。

# 线程池与消息队列

1、两者内部都使用了队列，如阻塞队列、优先级队列；

2、使用线程池时应用服务器既充当生产者又充当消费者，也是消息队列中间件的实现者，使用消息队列时中间件、生产者、消费者可以部署在不同的应用机器上（当然也可以部署在一台服务器上但很少有人这么用）；

3、出于第2点线程池更适合非分布式的系统，但在分布式架构下消息队列明显是更优项；

4、使用消息队列会带来额外的网络开销；

5、消息队列的耦合性更低，可扩展性更好，适用于弱一致性的场景，如对log日志的解耦；

6、消息队列自动实现消息的持久化，中间已经实现了大量功能，如消息转发、消息拒绝、消息重试，以及对消息的一些监控，例如消息的消费状态、消息的消费速率等，使用线程池如果需要很多功能还要自己去实现，例如需要执行状态需要打印队列数量、计算消息消费速度；

7、在不同系统间的服务调用（调用协议也可能不一致）线程池很难实现或开销很大，这时候消息队列可以屏蔽不同机器或不同协议的问题；

8、使用消息队列会提升系统的复杂度，网络抖动怎么办？最大队列长度怎么设置？超时时间又设置多少？QOS又设置为多少？消费者多少个比较合适？Channel cache size又该设置为多少？业务线可能都是用同一个MQ，你占资源太多，或者设计不当可能会导致整个MQ故障。

# 对比

## 资料文档

Kafka：中。有Kafka作者自己写的书，网上资料也有一些。

RabbitMQ：多。有一些不错的书，网上资料多。

ZeroMQ：少。没有专门写ZeroMQ的书，网上的资料多是一些代码的实现和简单介绍。

RocketMQ：少。没有专门写RocketMQ的书，网上的资料良莠不齐，官方文档很简洁，但是对技术细节没有过多的描述。

ActiveMQ：多。没有专门写ActiveMQ的书，网上资料多。

## 开发语言

Kafka：Scala

RabbitMQ：Erlang

ZeroMQ：C

RocketMQ：java

ActiveMQ：java

## 支持的协议

Kafka：自己定义的一套（基于TCP）

RabbitMQ：AMQP

ZeroMQ：TCP、UDP

RocketMQ：自己定义的一套

ActiveMQ：OpenWire、STOMP、REST、XMPP、AMQP

## 消息事务

Kafka：支持

RabbitMQ：支持。客户端将信道设置为事务模式，只有当消息被RabbitMQ接收，事务才能提交成功，否则在捕获异常后进行回滚。使用事务会使得性能有所下降

ZeroMQ：不支持

RocketMQ：支持

ActiveMQ：支持

## 负载均衡

Kafka：支持负载均衡。

1、一个broker通常就是一台服务器节点。对于同一个Topic的不同分区，Kafka会尽力将这些分区分布到不同的Broker服务器上，zookeeper保存了broker、主题和分区的元数据信息。分区首领会处理来自客户端的生产请求，Kafka分区首领会被分配到不同的broker服务器上，让不同的broker服务器共同分担任务。

每一个broker都缓存了元数据信息，客户端可以从任意一个broker获取元数据信息并缓存起来，根据元数据信息知道要往哪里发送请求。

2、Kafka的消费者组订阅同一个topic，会尽可能地使得每一个消费者分配到相同数量的分区，分摊负载。

3、当消费者加入或者退出消费者组的时候，还会触发再均衡，为每一个消费者重新分配分区，分摊负载。

Kafka的负载均衡大部分是自动完成的，分区的创建也是Kafka完成的，隐藏了很多细节，避免了繁琐的配置和人为疏忽造成的负载问题。

4、发送端由topic和key来决定消息发往哪个分区，如果key为null，那么会使用轮询算法将消息均衡地发送到同一个topic的不同分区中。如果key不为null，那么会根据key的hashcode取模计算出要发往的分区。

RabbitMQ：对负载均衡的支持不好。

1、消息被投递到哪个队列是由交换器和key决定的，交换器、路由键、队列都需要手动创建。

RabbitMQ客户端发送消息要和broker建立连接，需要事先知道broker上有哪些交换器，有哪些队列。通常要声明要发送的目标队列，如果没有目标队列，会在broker上创建一个队列，如果有，就什么都不处理，接着往这个队列发送消息。假设大部分繁重任务的队列都创建在同一个broker上，那么这个broker的负载就会过大。（可以在上线前预先创建队列，无需声明要发送的队列，但是发送时不会尝试创建队列，可能出现找不到队列的问题，RabbitMQ的备份交换器会把找不到队列的消息保存到一个专门的队列中，以便以后查询使用）

使用镜像队列机制建立RabbitMQ集群可以解决这个问题，形成master-slave的架构，master节点会均匀分布在不同的服务器上，让每一台服务器分摊负载。slave节点只是负责转发，在master失效时会选择加入时间最长的slave成为master。

当新节点加入镜像队列的时候，队列中的消息不会同步到新的slave中，除非调用同步命令，但是调用命令后，队列会阻塞，不能在生产环境中调用同步命令。

2、当RabbitMQ队列拥有多个消费者的时候，队列收到的消息将以轮询的分发方式发送给消费者。每条消息只会发送给订阅列表里的一个消费者，不会重复。

这种方式非常适合扩展，而且是专门为并发程序设计的。

如果某些消费者的任务比较繁重，那么可以设置basicQos限制信道上消费者能保持的最大未确认消息的数量，在达到上限时，RabbitMQ不再向这个消费者发送任何消息。

3、对于RabbitMQ而言，客户端与集群建立的TCP连接不是与集群中所有的节点建立连接，而是挑选其中一个节点建立连接。

但是RabbitMQ集群可以借助HAProxy、LVS技术，或者在客户端使用算法实现负载均衡，引入负载均衡之后，各个客户端的连接可以分摊到集群的各个节点之中。

客户端均衡算法：

1)轮询法。按顺序返回下一个服务器的连接地址。

2)加权轮询法。给配置高、负载低的机器配置更高的权重，让其处理更多的请求；而配置低、负载高的机器，给其分配较低的权重，降低其系统负载。

3)随机法。随机选取一个服务器的连接地址。

4)加权随机法。按照概率随机选取连接地址。

5)源地址哈希法。通过哈希函数计算得到的一个数值，用该数值对服务器列表的大小进行取模运算。

6)最小连接数法。动态选择当前连接数最少的一台服务器的连接地址。

ZeroMQ：去中心化，不支持负载均衡。本身只是一个多线程网络库。

RocketMQ：支持负载均衡。

一个broker通常是一个服务器节点，broker分为master和slave,master和slave存储的数据一样，slave从master同步数据。

1、nameserver与每个集群成员保持心跳，保存着Topic-Broker路由信息，同一个topic的队列会分布在不同的服务器上。

2、发送消息通过轮询队列的方式发送，每个队列接收平均的消息量。发送消息指定topic、tags、keys，无法指定投递到哪个队列（没有意义，集群消费和广播消费跟消息存放在哪个队列没有关系）。

tags选填，类似于 Gmail 为每封邮件设置的标签，方便服务器过滤使用。目前只支 持每个消息设置一个 tag，所以也可以类比为 Notify 的 MessageType 概念。

keys选填，代表这条消息的业务关键词，服务器会根据 keys 创建哈希索引，设置后， 可以在 Console 系统根据 Topic、Keys 来查询消息，由于是哈希索引，请尽可能 保证 key 唯一，例如订单号，商品 Id 等。

3、RocketMQ的负载均衡策略规定：Consumer数量应该小于等于Queue数量，如果Consumer超过Queue数量，那么多余的Consumer 将不能消费消息。这一点和Kafka是一致的，RocketMQ会尽可能地为每一个Consumer分配相同数量的队列，分摊负载。

ActiveMQ：支持负载均衡。可以基于zookeeper实现负载均衡。

## 集群方式

Kafka：天然的‘Leader-Slave’无状态集群，每台服务器既是Master也是Slave。

分区首领均匀地分布在不同的Kafka服务器上，分区副本也均匀地分布在不同的Kafka服务器上，所以每一台Kafka服务器既含有分区首领，同时又含有分区副本，每一台Kafka服务器是某一台Kafka服务器的Slave，同时也是某一台Kafka服务器的leader。

Kafka的集群依赖于zookeeper，zookeeper支持热扩展，所有的broker、消费者、分区都可以动态加入移除，而无需关闭服务，与不依靠zookeeper集群的mq相比，这是最大的优势。

RabbitMQ：支持简单集群，'复制'模式，对高级集群模式支持不好。

RabbitMQ的每一个节点，不管是单一节点系统或者是集群中的一部分，要么是内存节点，要么是磁盘节点，集群中至少要有一个是磁盘节点。

在RabbitMQ集群中创建队列，集群只会在单个节点创建队列进程和完整的队列信息（元数据、状态、内容），而不是在所有节点上创建。

引入镜像队列，可以避免单点故障，确保服务的可用性，但是需要人为地为某些重要的队列配置镜像。

ZeroMQ：去中心化，不支持集群。

RocketMQ：常用 多对'Master-Slave' 模式，开源版本需手动切换Slave变成Master

Name Server是一个几乎无状态节点，可集群部署，节点之间无任何信息同步。

Broker部署相对复杂，Broker分为Master与Slave，一个Master可以对应多个Slave，但是一个Slave只能对应一个Master，Master与Slave的对应关系通过指定相同的BrokerName，不同的BrokerId来定义，BrokerId为0表示Master，非0表示Slave。Master也可以部署多个。每个Broker与Name Server集群中的所有节点建立长连接，定时注册Topic信息到所有Name Server。

Producer与Name Server集群中的其中一个节点（随机选择）建立长连接，定期从Name Server取Topic路由信息，并向提供Topic服务的Master建立长连接，且定时向Master发送心跳。Producer完全无状态，可集群部署。

Consumer与Name Server集群中的其中一个节点（随机选择）建立长连接，定期从Name Server取Topic路由信息，并向提供Topic服务的Master、Slave建立长连接，且定时向Master、Slave发送心跳。Consumer既可以从Master订阅消息，也可以从Slave订阅消息，订阅规则由Broker配置决定。

客户端先找到NameServer, 然后通过NameServer再找到 Broker。

一个topic有多个队列，这些队列会均匀地分布在不同的broker服务器上。RocketMQ队列的概念和Kafka的分区概念是基本一致的，Kafka同一个topic的分区尽可能地分布在不同的broker上，分区副本也会分布在不同的broker上。

RocketMQ集群的slave会从master拉取数据备份，master分布在不同的broker上。

ActiveMQ：支持简单集群模式，比如'主-备'，对高级集群模式支持不好。

## 管理界面

Kafka：一般

RabbitMQ：好

ZeroMQ：无

RocketMQ：无

ActiveMQ：一般

## 可用性

Kafka：非常高（分布式）

RabbitMQ：高（主从）

ZeroMQ：高。

RocketMQ：非常高（分布式）

ActiveMQ：高（主从）

## 吞吐量TPS

Kafka：极大 Kafka按批次发送消息和消费消息。发送端将多个小消息合并，批量发向Broker，消费端每次取出一个批次的消息批量处理。

RabbitMQ：比较大

ZeroMQ：极大

RocketMQ：大 RocketMQ接收端可以批量消费消息，可以配置每次消费的消息数，但是发送端不是批量发送。

ActiveMQ：比较大

## 并发度

Kafka：高

一个线程一个消费者，Kafka限制消费者的个数要小于等于分区数，如果要提高并行度，可以在消费者中再开启多线程，或者增加consumer实例数量。

RabbitMQ：极高

本身是用Erlang语言写的，并发性能高。

可在消费者中开启多线程，最常用的做法是一个channel对应一个消费者，每一个线程把持一个channel，多个线程复用connection的tcp连接，减少性能开销。

当RabbitMQ队列拥有多个消费者的时候，队列收到的消息将以轮询的分发方式发送给消费者。每条消息只会发送给订阅列表里的一个消费者，不会重复。

这种方式非常适合扩展，而且是专门为并发程序设计的。

如果某些消费者的任务比较繁重，那么可以设置basicQos限制信道上消费者能保持的最大未确认消息的数量，在达到上限时，RabbitMQ不再向这个消费者发送任何消息。

ZeroMQ：高

RocketMQ：高

1、RocketMQ限制消费者的个数少于等于队列数，但是可以在消费者中再开启多线程，这一点和Kafka是一致的，提高并行度的方法相同。

修改消费并行度方法

a) 同一个 ConsumerGroup 下，通过增加 Consumer 实例数量来提高并行度，超过订阅队列数的 Consumer实例无效。

b) 提高单个 Consumer 的消费并行线程，通过修改参数consumeThreadMin、consumeThreadMax

2、同一个网络连接connection，客户端多个线程可以同时发送请求，连接会被复用，减少性能开销。

ActiveMQ：高

单个ActiveMQ的接收和消费消息的速度在1万笔/秒（持久化 一般为1-2万， 非持久化 2 万以上），在生产环境中部署10个ActiveMQ就能达到10万笔/秒以上的性能，部署越多的ActiveMQ broker 在MQ上latency也就越低，系统吞吐量也就越高。

## 消息存储

Kafka：内存、磁盘、数据库。支持大量堆积。

Kafka的最小存储单元是分区，一个topic包含多个分区，Kafka创建主题时，这些分区会被分配在多个服务器上，通常一个broker一台服务器。 分区首领会均匀地分布在不同的服务器上，分区副本也会均匀的分布在不同的服务器上，确保负载均衡和高可用性，当新的broker加入集群的时候，部分副本会被移动到新的broker上。 根据配置文件中的目录清单，Kafka会把新的分区分配给目录清单里分区数最少的目录。 默认情况下，分区器使用轮询算法把消息均衡地分布在同一个主题的不同分区中，对于发送时指定了key的情况，会根据key的hashcode取模后的值存到对应的分区中。

RabbitMQ：内存、磁盘。支持少量堆积。

RabbitMQ的消息分为持久化的消息和非持久化消息，不管是持久化的消息还是非持久化的消息都可以写入到磁盘。 持久化的消息在到达队列时就写入到磁盘，并且如果可以，持久化的消息也会在内存中保存一份备份，这样可以提高一定的性能，当内存吃紧的时候会从内存中清除。非持久化的消息一般只存在于内存中，在内存吃紧的时候会被换入到磁盘中，以节省内存。

引入镜像队列机制，可将重要队列“复制”到集群中的其他broker上，保证这些队列的消息不会丢失。配置镜像的队列，都包含一个主节点master和多个从节点slave,如果master失效，加入时间最长的slave会被提升为新的master，除发送消息外的所有动作都向master发送，然后由master将命令执行结果广播给各个slave，RabbitMQ会让master均匀地分布在不同的服务器上，而同一个队列的slave也会均匀地分布在不同的服务器上，保证负载均衡和高可用性。

ZeroMQ：消息发送端的内存或者磁盘中。不支持持久化。

RocketMQ：磁盘。支持大量堆积。

commitLog文件存放实际的消息数据，每个commitLog上限是1G，满了之后会自动新建一个commitLog文件保存数据。ConsumeQueue队列只存放offset、size、tagcode，非常小，分布在多个broker上。ConsumeQueue相当于CommitLog的索引文件，消费者消费时会从consumeQueue中查找消息在commitLog中的offset，再去commitLog中查找元数据。

ConsumeQueue存储格式的特性，保证了写过程的顺序写盘（写CommitLog文件），大量数据IO都在顺序写同一个commitLog，满1G了再写新的。加上RocketMQ是累计4K才强制从PageCache中刷到磁盘（缓存），所以高并发写性能突出。

ActiveMQ：内存、磁盘、数据库。支持少量堆积。

## 订阅形式和消息分发

Kafka：基于topic以及按照topic进行正则匹配的发布订阅模式。

【发送】

发送端由topic和key来决定消息发往哪个分区，如果key为null，那么会使用轮询算法将消息均衡地发送到同一个topic的不同分区中。如果key不为null，那么会根据key的hashcode取模计算出要发往的分区。

【接收】

1>consumer向群组协调器broker发送心跳来维持他们和群组的从属关系以及他们对分区的所有权关系，所有权关系一旦被分配就不会改变除非发生再均衡(比如有一个consumer加入或者离开consumer group)，consumer只会从对应的分区读取消息。

2>Kafka限制consumer个数要少于分区个数,每个消息只会被同一个 Consumer Group的一个consumer消费（非广播）。

3>Kafka的 Consumer Group订阅同一个topic，会尽可能地使得每一个consumer分配到相同数量的分区，不同 Consumer Group订阅同一个主题相互独立，同一个消息会被不同的 Consumer Group处理。

RabbitMQ：提供了4种：direct, topic ,Headers和fanout。

【发送】

先要声明一个队列，这个队列会被创建或者已经被创建，队列是基本存储单元。

由exchange和key决定消息存储在哪个队列。

direct>发送到和bindingKey完全匹配的队列。

topic>路由key是含有"."的字符串，会发送到含有“\*”、“#”进行模糊匹配的bingKey对应的队列。

fanout>与key无关，会发送到所有和exchange绑定的队列

headers>与key无关，消息内容的headers属性（一个键值对）和绑定键值对完全匹配时，会发送到此队列。此方式性能低一般不用

【接收】

RabbitMQ的队列是基本存储单元，不再被分区或者分片，对于我们已经创建了的队列，消费端要指定从哪一个队列接收消息。

当RabbitMQ队列拥有多个消费者的时候，队列收到的消息将以轮询的分发方式发送给消费者。每条消息只会发送给订阅列表里的一个消费者，不会重复。

这种方式非常适合扩展，而且是专门为并发程序设计的。

如果某些消费者的任务比较繁重，那么可以设置basicQos限制信道上消费者能保持的最大未确认消息的数量，在达到上限时，RabbitMQ不再向这个消费者发送任何消息。

ZeroMQ：点对点(p2p)

RocketMQ：基于topic/messageTag以及按照消息类型、属性进行正则匹配的发布订阅模式

【发送】

发送消息通过轮询队列的方式发送，每个队列接收平均的消息量。发送消息指定topic、tags、keys，无法指定投递到哪个队列（没有意义，集群消费和广播消费跟消息存放在哪个队列没有关系）。

tags选填，类似于 Gmail 为每封邮件设置的标签，方便服务器过滤使用。目前只支 持每个消息设置一个 tag，所以也可以类比为 Notify 的 MessageType 概念。

keys选填，代表这条消息的业务关键词，服务器会根据 keys 创建哈希索引，设置后， 可以在 Console 系统根据 Topic、Keys 来查询消息，由于是哈希索引，请尽可能 保证 key 唯一，例如订单号，商品 Id 等。

【接收】

1>广播消费。一条消息被多个Consumer消费，即使Consumer属于同一个ConsumerGroup，消息也会被ConsumerGroup中的每个Consumer都消费一次。

2>集群消费。一个 Consumer Group中的Consumer实例平均分摊消费消息。例如某个Topic有 9 条消息，其中一个Consumer Group有3个实例，那么每个实例只消费其中的 3 条消息。即每一个队列都把消息轮流分发给每个consumer。

ActiveMQ：点对点(p2p)、广播（发布-订阅）

点对点模式，每个消息只有1个消费者；

发布/订阅模式，每个消息可以有多个消费者。

【发送】

点对点模式：先要指定一个队列，这个队列会被创建或者已经被创建。

发布/订阅模式：先要指定一个topic，这个topic会被创建或者已经被创建。

【接收】

点对点模式：对于已经创建了的队列，消费端要指定从哪一个队列接收消息。

发布/订阅模式：对于已经创建了的topic，消费端要指定订阅哪一个topic的消息。

## 顺序消息

Kafka：支持。

设置生产者的max.in.flight.requests.per.connection为1，可以保证消息是按照发送顺序写入服务器的，即使发生了重试。

Kafka保证同一个分区里的消息是有序的，但是这种有序分两种情况

1、key为null，消息逐个被写入不同主机的分区中，但是对于每个分区依然是有序的

2、key不为null , 消息被写入到同一个分区，这个分区的消息都是有序。

RabbitMQ：不支持

ZeroMQ：不支持

RocketMQ：支持

ActiveMQ：不支持

## 消息确认

Kafka：支持。

1、发送方确认机制

ack=0，不管消息是否成功写入分区

ack=1，消息成功写入首领分区后，返回成功

ack=all，消息成功写入所有分区后，返回成功。

2、接收方确认机制

自动或者手动提交分区偏移量，早期版本的Kafka偏移量是提交给Zookeeper的，这样使得zookeeper的压力比较大，更新版本的Kafka的偏移量是提交给Kafka服务器的，不再依赖于zookeeper群组，集群的性能更加稳定。

RabbitMQ：支持。

1、发送方确认机制，消息被投递到所有匹配的队列后，返回成功。如果消息和队列是可持久化的，那么在写入磁盘后，返回成功。支持批量确认和异步确认。

2、接收方确认机制，设置autoAck为false，需要显式确认，设置autoAck为true，自动确认。

当autoAck为false的时候，RabbitMQ队列会分成两部分，一部分是等待投递给consumer的消息，一部分是已经投递但是没收到确认的消息。如果一直没有收到确认信号，并且consumer已经断开连接，RabbitMQ会安排这个消息重新进入队列，投递给原来的消费者或者下一个消费者。

未确认的消息不会有过期时间，如果一直没有确认，并且没有断开连接，RabbitMQ会一直等待，RabbitMQ允许一条消息处理的时间可以很久很久。

ZeroMQ：支持。

RocketMQ：支持。

ActiveMQ：支持。

## 消息回溯

Kafka：支持指定分区offset位置的回溯。

RabbitMQ：不支持

ZeroMQ：不支持

RocketMQ：支持指定时间点的回溯。

ActiveMQ：不支持

## 消息重复

Kafka：支持at least once、at most once

RabbitMQ：支持at least once、at most once

ZeroMQ：只有重传机制，但是没有持久化，消息丢了重传也没有用。既不是at least once、也不是at most once、更不是exactly only once

RocketMQ：支持at least once

ActiveMQ：支持at least once

## 消息重试

Kafka：不支持，但是可以实现。

Kafka支持指定分区offset位置的回溯，可以实现消息重试。

RabbitMQ：不支持，但是可以利用消息确认机制实现。

RabbitMQ接收方确认机制，设置autoAck为false。

当autoAck为false的时候，RabbitMQ队列会分成两部分，一部分是等待投递给consumer的消息，一部分是已经投递但是没收到确认的消息。如果一直没有收到确认信号，并且consumer已经断开连接，RabbitMQ会安排这个消息重新进入队列，投递给原来的消费者或者下一个消费者。

ZeroMQ：不支持，

RocketMQ：支持。

消息消费失败的大部分场景下，立即重试99%都会失败，所以RocketMQ的策略是在消费失败时定时重试，每次时间间隔相同。

1、发送端的 send 方法本身支持内部重试，重试逻辑如下：

a)至多重试3次；

b)如果发送失败，则轮转到下一个broker；

c)这个方法的总耗时不超过sendMsgTimeout 设置的值，默认 10s，超过时间不在重试。

2、接收端。

Consumer 消费消息失败后，要提供一种重试机制，令消息再消费一次。Consumer 消费消息失败通常可以分为以下两种情况：

由于消息本身的原因，例如反序列化失败，消息数据本身无法处理（例如话费充值，当前消息的手机号被

注销，无法充值）等。定时重试机制，比如过 10s 秒后再重试。

由于依赖的下游应用服务不可用，例如 db 连接不可用，外系统网络不可达等。

即使跳过当前失败的消息，消费其他消息同样也会报错。这种情况可以 sleep 30s，再消费下一条消息，减轻 Broker 重试消息的压力。

ActiveMQ：不支持