# 什么是消息中间件

消息中间件是一套系统（或平台），用于应用程序之间进行通信，系统通过消息传递完成交互。

消息中间件的主要特点有以下几个。

1. 分布式：消息中间件都是分布式的，因此才可以提供异步、解耦等功能。

2. 可靠性：基于消息的通信是可靠的，消息不会丢失。大多数消息中间件都提供将消息持久化到磁盘的功能。

3. 异步：通过消息中间件，可将远程同步调用拆解成为异步调用。对于不需要获取远程调用结果的应用场景来说，性能提升明显。

4. 松耦合：消息直接由中间件存储和分发。消息生产者只需关注如何将消息发送给消息中介服务器；消费者只需关注如何从中介服务器订阅。生产者和消费者之间是完全解耦的，不需要知道彼此的存在。

5. 事件驱动：可以将复杂的应用系统重构成为事件驱动的系统。事件溯源（Event Sourcing），表示一个对象从创建到消亡，会经过的多种状态。如果把对象的状态变化都存储下来，不但可以根据状态变化记录获取对象的当前状态，也可以回溯对象的变化过程。消息中间件能很好地支持这样的系统设计方式，将触发对象状态变化的事件放入消息队列。

在带来好处的同时，引入消息中间件也有一些需要注意的地方。

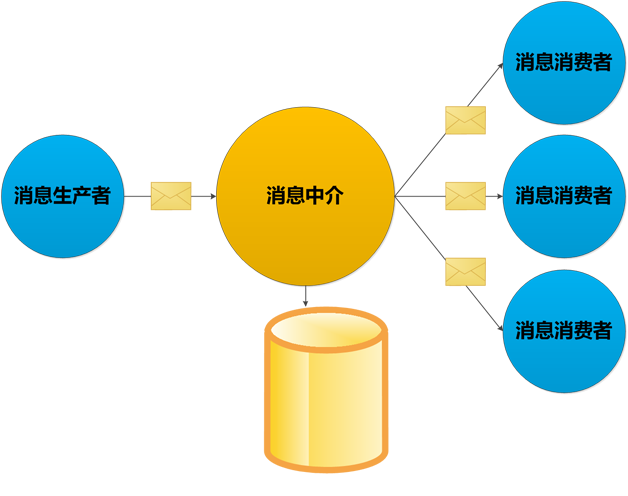
1. 分布式带来的复杂性：消息中间件都是分布式的，引入分布式会大大增加系统复杂度，在不同主机、不同进程之间的调用和调试，会带来更多的不稳定性。分布式系统还会增加对外部系统的依赖。即使自己的系统没有问题，也可能会因为依赖系统出问题而导致系统不稳定。因此，Martin Fowler曾说：“分布式调用的第一原则就是不要分布式。”

2. 同步调用应该考虑其他方式：尽管消息中间件也可用于同步调用，但这并不是它的长项，同步调用可以考虑使用HTTP、NIO等其他方式。

图1描述了消息中间件的组成部分。

3. 消息中介（Broker）：可理解为消息中间件的服务器。消息中介用于存储消息，并且维护消息消费者和消息队列之间的订阅关系（也可由消费者自己维护）。

消息在中介如何存储，是决定消息中间件功能和性能的最重要因素。目前来说，最主要的两种存储消息的方式是kv存储和顺序存储。后文将详细介绍两种不同存储引擎消息中间件的区别。



1. 消息生产者（Producer）：与消息消费者一起组成消息中间件的客户端。生产者用于发送消息到消息中介。

客户端连接服务器一般可以选择TCP、HTTP等协议。内网基于长连接的TCP协议效率更高，公网可以考虑HTTP协议穿透防火墙。

2. 消息消费者（Consumer）：与消息生产者一起组成消息中间件的客户端。消费者用于从消息中介获得消息并交给业务系统使用。

消息的消费分成推送和拉取两种模式。推送是消息中介主动将消息发送给消息消费者，拉取则是消息消费者主动从消息中介获取消息。两种模式的使用场景不太一样，各有优缺点，下文也会详细介绍。

为了便于理解，在这里将消息中间件和关系型数据库做一个比较：

1. 消息中介相当于数据库的服务器；

2. 消息生产者相当于使用INSERT语句的SQL客户端；

3. 消息消费者相当于使用SELECT语句的SQL客户端。

当然这个比较不是非常恰当。例如根据实现方式不同，消息的删除可能由消费者发起，也可能由消息中介主动发起。但能比较直观说明，消息中间件是由服务器和客户端组成，以及它们所承担的职责。

**JMS**

JMS的全称是Java Message Service，即Java消息服务，定义了Java平台消息中间件的技术规范。JMS只提供了应用程序对消息中间件操作的接口规范，并未提供实现，其实现由各个消息中间件厂商的驱动程序来提供，和Java的另一个规范标准JDBC相似。遵循JMS规范的消息中间件都使用统一的API。

JMS定义了消息的编程模型，如连接工厂、会话、消息目的地、消息生产者、消息消费者、消息体、消息优先级和消息类型等。本文并不会详细介绍JMS，有兴趣了解请查阅相关资料。但会重点介绍JMS中定义的消息类型，因为后面介绍的几种消息中间件如何支持消息类型是本文的关键之一。

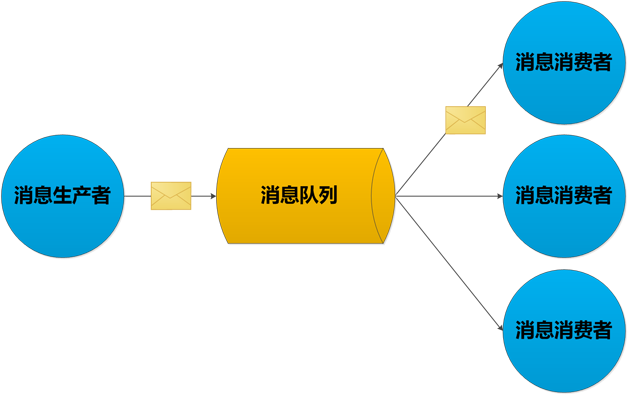
消息类型分为点对点和发布/订阅两种。

1. 点对点（Point To Point）：消息生产者将消息发送到消息队列（Queue）中，只有一个消费者能够消费此消息，消费完成之后消息即删除。

这里应该注意的是，任意一个消费者都可以消费这个消息，但消息绝对不会被两个消费者重复消费。

消息的消费者和生产者没有时间依赖，可以先发送消息，再启动消费者。

图2展示了JMS的点对点消息类型。

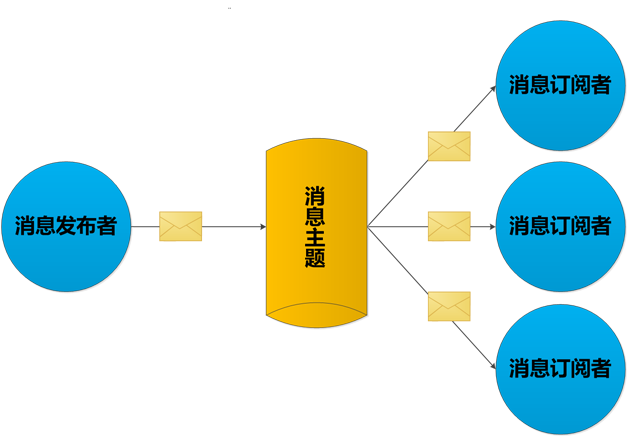


2. 发布/订阅（Publish/Subscribe）：消息生产者将消息发送到消息主题（Topic）中，所有订阅这个主题的消费者都可以消费此消息，当所有订阅者都消费完成之后才能删除消息。

消息的生产者和消费者之间有时间依赖，只有事先订阅这个主题的消费者才可消费。如果先发送消息，后订阅主题，那么订阅之前的消息将不能被这个订阅者消费。

订阅者又可分为持久化订阅和非持久化订阅。如果持久化的订阅者在订阅之后离线，收到的消息仍会在订阅者再次上线时收到，不会错过消息。而非持久化的订阅者一旦离线，离线时的消息将被错过。

图3展示了JMS的发布/订阅消息类型。



最后，需要介绍消息传递语义（Message Delivery Semantics）。消息传递的担保有3种级别：最多一次（At Most Once），至少一次（At Least Once）和精确地仅发送一次（Exactly Once）。

【最多一次】

消息只发送或消费一次，无论消息中介是否收到消息，或者消息是否已消费成功，都不会再次发送。

这样做的问题是，消息可能会丢失。虽然客户端发送了消息，但消息中介还来不及存储就崩溃，那么这条消息就会丢失。

【至少一次】

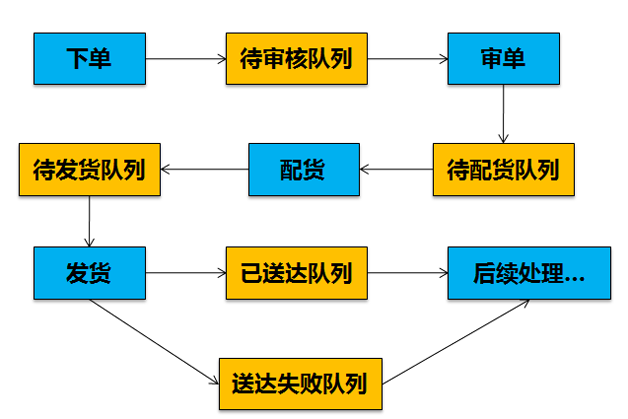
如果消息中介没有明确告诉客户端消息已经收到，那么客户端会重新发送或消费这条消息。

这样做的问题是，消息可能会重复发送或消费。例如消息已经存到了消息中介，但还没来得及给客户端发送确认信息，消息中介就崩溃了。那么等消息中介重新启动之后，客户端会重新发送这条消息，造成重复。

【精确地仅发送一次】

既不会多也不会少发送消息。一般通过事务来实现，只有消息中介收到客户端确认处理成功的信息，才会提交事务，否则在经过一定时间限制之后消息会回滚。精确地仅发送一次消息，不会丢失也不会有重复的消息，但要达到这一点，对性能的损耗非常大。

JMS要求消息精确地仅发送一次。



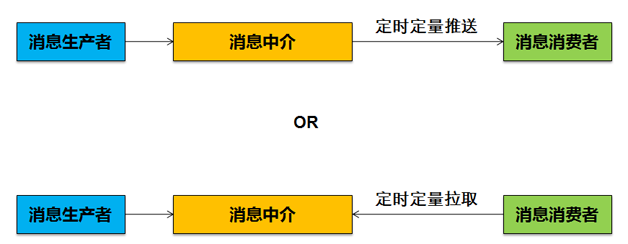
应用场景

【业务解耦】

各个业务系统仅需要处理自己的业务逻辑，并且发送事件消息到消息中间件。下游业务系统直接订阅消息中间件的队列或主题获取事件。这样不但解耦了系统间的依赖，而且使调用异步化，提升了系统的性能。图4显示了如何通过消息中间件将订单系统解耦。

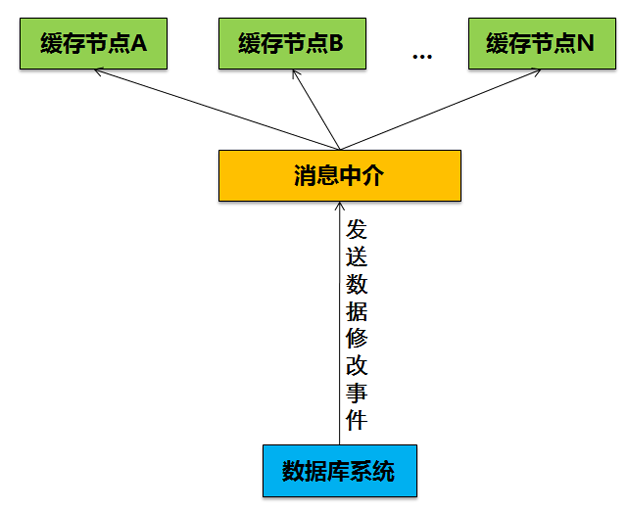
【削峰填谷】

如果上游系统的吞吐能力强于下游系统，那么在上游系统满负荷时将冲垮下游系统。使用消息中间件的定时定量推送或者定时定量拉取，可在上游峰值时堆积消息，在峰值过去时慢慢消费，增强系统的缓冲能力。图5显示了如何通过消息中间件缓冲业务洪峰，并且使用定时定量分流消费消息。



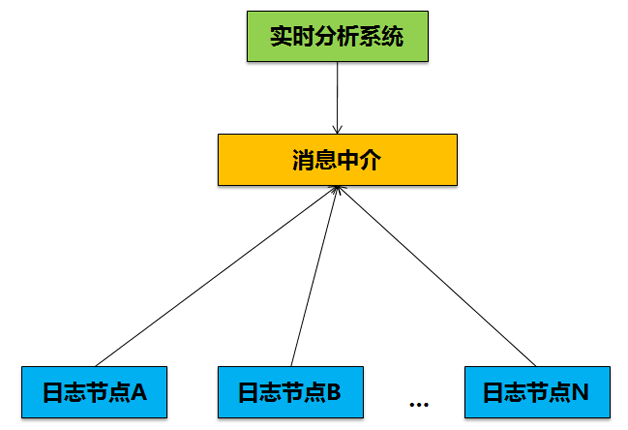
【广播通知】

系统一个状态的改变，需要通知多个相关系统，可通过消息订阅的方式推送给各个订阅者系统。比如数据库值的改变，需要通知所有的缓存系统更新，可以把数据库值改变发送消息给消息中间件，然后各缓存订阅相关主题，收到消息后更新自己的缓存。图6显示了数据中心如何通过广播通知已订阅的缓存系统更新数据。



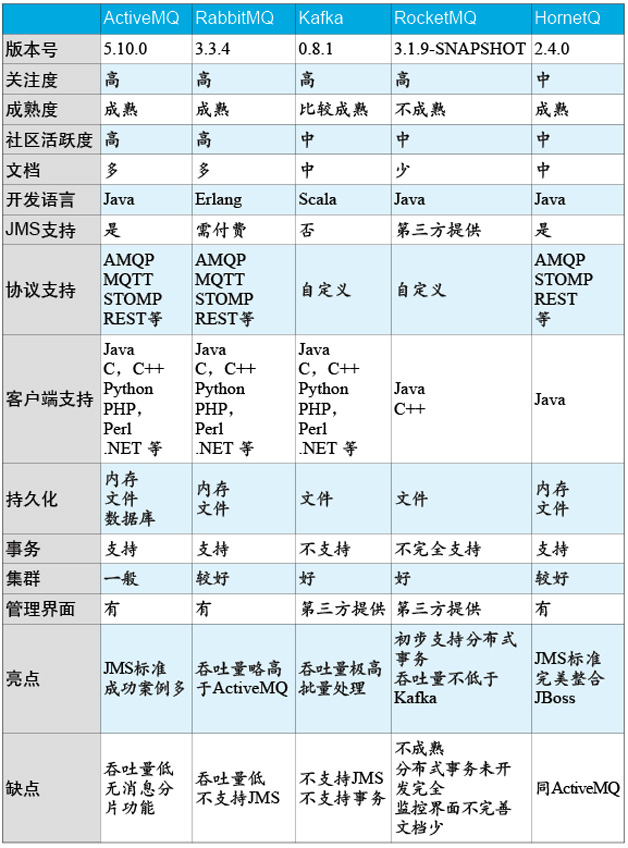
【日志分析】

日志分析往往需要处理大量日志，不可能存储在一台物理机上。消息中间件可提供一个集群，用来存储海量消息，将其缓存到消息中间件。比较常用的日志分析系统是使用日志收集组件（如Flume）收集，存储到高吞吐量的消息中间件（如Kafka），供实时分析系统（如Storm）分析日志。图7描述了日志分析系统的基本组成部分。



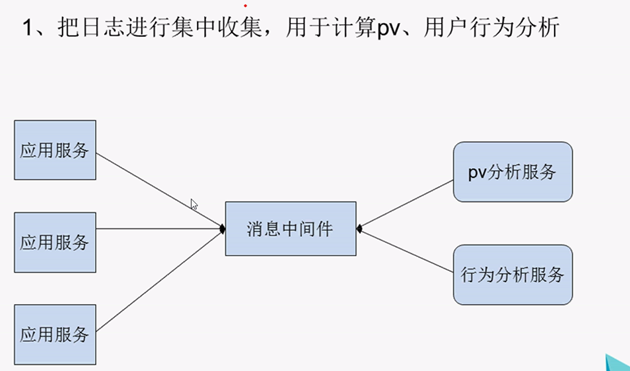
消息中间件概览

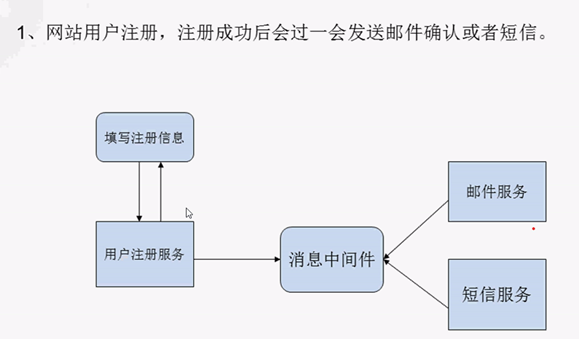
表1展示了目前比较流行的消息中间件的简单对比，希望对读者的选型有所帮助。

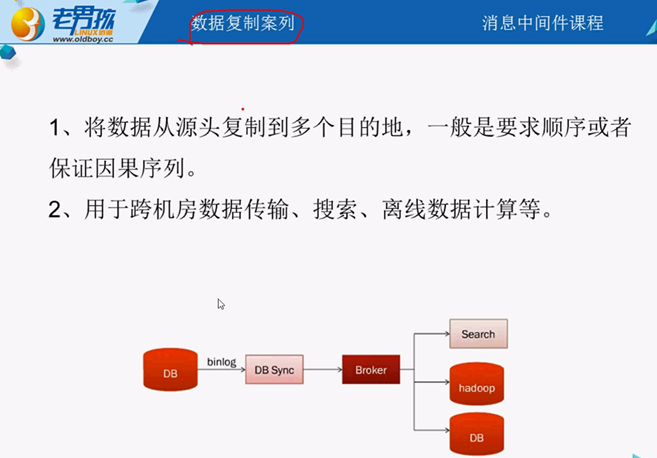


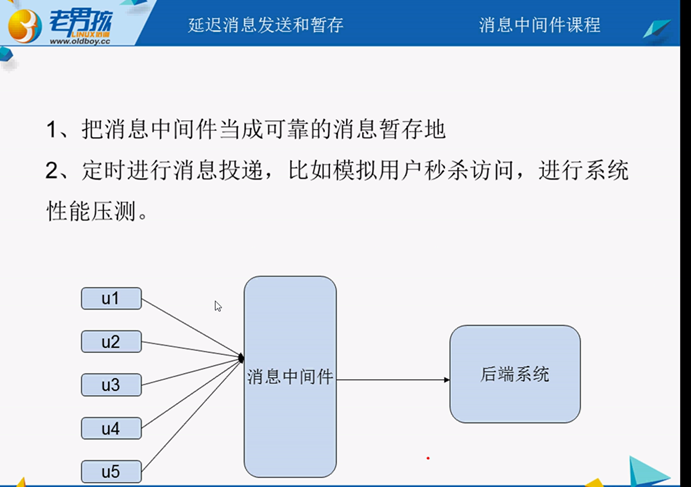
转自<http://m.blog.csdn.net/article/details?id=51868006>

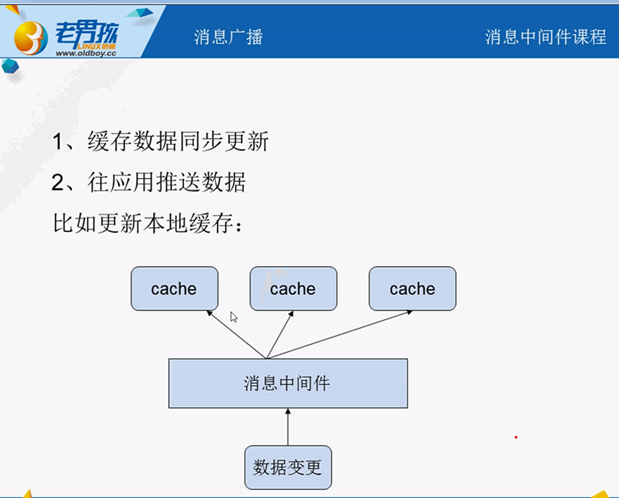
# 消息中间件的多个企业级应用场景











# Zookper工作原理

<http://gotowqj.iteye.com/blog/2065572>

# 消息中间件之zookper安装部署

[root@q tools]# chmod 755 jdk-8u40-linux-x64.rpm

[root@q tools]# rpm -i jdk-8u40-linux-x64.rpm

Unpacking JAR files...

rt.jar...

jsse.jar...

charsets.jar...

tools.jar...

localedata.jar...

jfxrt.jar...

plugin.jar...

javaws.jar...

deploy.jar...

[root@q tools]# java -version #检查版本

java version "1.8.0\_40"

Java(TM) SE Runtime Environment (build 1.8.0\_40-b25)

Java HotSpot(TM) 64-Bit Server VM (build 25.40-b25, mixed mode)

[root@q tools]# mkdir zk1 zk2 zk3

[root@q tools]# cd zk1/

[root@q zk1]# ls

zookeeper-3.4.6.tar.gz

[root@q zk1]# tar xf zookeeper-3.4.6\_.tar.gz

zookeeper-3.4.6 zookeeper-3.4.6\_.tar.gz

[root@q zk1]# cd zookeeper-3.4.6

创建日志目录和数据存放的目录

[root@q zookeeper-3.4.6]# mkdir dataDir dataLogDir

[root@q zookeeper-3.4.6]# cd dataDir/

[root@q dataDir]# touch myid #创建一个myid的文件

[root@q dataDir]# echo "1" >> myid #编号为1

[root@q dataDir]# cd ..

[root@q zookeeper-3.4.6]# cd conf/

[root@q conf]# ls

configuration.xsl log4j.properties zoo\_sample.cfg

[root@q conf]# mv zoo\_sample.cfg zoo.cfg #修改为zoo.cfg

[root@q conf]# ls

configuration.xsl log4j.properties zoo.cfg

[root@q conf]# vim zoo.cfg

dataDir=/home/zsq/tools/zk1/zookeeper-3.4.6/dataDir

dataLogDir=/home/zsq/tools/zk1/zookeeper-3.4.6/dataLogDir

# the port at which the c lients will connect

clientPort=2181 #端口

server.1=127.0.0.1:8880:7770 #server.1的1是刚才创建的编号1，8880是选举的端口，7770是心跳检测的端口

server.2=127.0.0.1:8881:7771 #另外两台中的一台

server.3=127.0.0.1:8882:7772

[root@q tools]# cd zk1/

[root@q zk1]# ls

zookeeper-3.4.6 zookeeper-3.4.6\_.tar.gz

[root@q zk1]# cp -rf zookeeper-3.4.6 /home/zsq/tools/zk2

[root@q zk1]# cp -rf zookeeper-3.4.6 /home/zsq/tools/zk3

修改zk2和zk3

[root@q dataDir]# pwd

/home/zsq/tools/zk2/zookeeper-3.4.6/dataDir

[root@q dataDir]# cat myid

2

dataDir=/home/zsq/tools/zk2/zookeeper-3.4.6/dataDir

dataLogDir=/home/zsq/tools/zk2/zookeeper-3.4.6/dataLogDir

# the port at which the clients will connect

clientPort=2182

server.1=127.0.0.1:8880:7770

server.2=127.0.0.1:8881:7771

server.3=127.0.0.1:8882:7772

[root@q dataDir]# pwd

/home/zsq/tools/zk3/zookeeper-3.4.6/dataDir

[root@q dataDir]# cat myid

3

dataDir=/home/zsq/tools/zk3/zookeeper-3.4.6/dataDir

dataLogDir=/home/zsq/tools/zk3/zookeeper-3.4.6/dataLogDir

# the port at which the clients will connect

clientPort=2183

server.1=127.0.0.1:8880:7770

server.2=127.0.0.1:8881:7771

server.3=127.0.0.1:8882:7772

启动 zk1 zk2 zk3

[root@q zookeeper-3.4.6]# cd bin/

[root@q bin]# ls # zkServer.cmd是windows下的运行程序

README.txt zkCleanup.sh zkCli.cmd zkCli.sh zkEnv.cmd zkEnv.sh zkServer.cmd zkServer.sh

[root@q bin]# ./zkServer.sh start #启动

JMX enabled by default

Using config: /home/zsq/tools/zk1/zookeeper-3.4.6/bin/../conf/zoo.cfg

Starting zookeeper ... STARTED

启动完成会在目录下生成zookeeper.out

[root@q bin]# cat

README.txt zkCleanup.sh zkCli.cmd zkCli.sh zkEnv.cmd zkEnv.sh zkServer.cmd zkServer.sh zookeeper.out

相应的命令启动另外两台，进入到相应的路径

[root@q zookeeper-3.4.6]# netstat -ntpl|grep "218"

tcp 0 0 :::2181 :::\* LISTEN 5727/java

tcp 0 0 :::2182 :::\* LISTEN 5673/java

tcp 0 0 :::2183 :::\* LISTEN 5640/java

连接测试，能连进去就表面OK

[root@q bin]# ./zkCli.sh -server 127.0.0.1:2181

WATCHER::

WatchedEvent state:SyncConnected type:None path:null

[zk: 127.0.0.1:2181(CONNECTED) 0] ls /

[zookeeper]

[zk: 127.0.0.1:2181(CONNECTED) 1] ls /zookeeper

[quota]

[zk: 127.0.0.1:2181(CONNECTED) 2] ls /zookeeper/quota

[]

[zk: 127.0.0.1:2181(CONNECTED) 3] get /zookeeper/quota

cZxid = 0x0

ctime = Thu Jan 01 08:00:00 CST 1970

mZxid = 0x0

mtime = Thu Jan 01 08:00:00 CST 1970

pZxid = 0x0

cversion = 0

dataVersion = 0

aclVersion = 0

ephemeralOwner = 0x0

dataLength = 0

numChildren = 0

[zk: 127.0.0.1:2181(CONNECTED) 4] quit

Quitting...

# 消息中间件metaq安装并注册到zookper集群

项目地址

<https://github.com/killme2008/Metamorphosis>

Memorphosis是一个消息中间件，它是linkedin开源MQ——kafka的Java版本，针对淘宝内部应用做了定制和优化。Metamorphosis的设计原则

* 消息都是持久的，保存在磁盘
* 吞吐量第一
* 消费状态保存在客户端
* 分布式，生产者、服务器和消费者都可分布

Metamorphosis的部署结构

[root@q tools]# tar xf metaq-server-1.4.6.2.tar.gz

[root@q tools]# cd taobao/

[root@q taobao]# ls

COPYING.txt metamorphosis-server-wrapper README.md

[root@q taobao]# cd metamorphosis-server-wrapper/

[root@q metamorphosis-server-wrapper]# ls

bin conf lib provided

其中bin目录包含启动脚本（目前只有linux下的启动脚本），conf下为配置文件，lib下为meta服务器的所有jar包,logs为日志所在目录。

[root@q metamorphosis-server-wrapper]# cd conf/

[root@q conf]# ls

async\_slave.properties jettyBroker.properties notifySlave.properties samsa\_master.properties timetunnel.properties

gregor\_slave.properties notifyadaper sample.server.ini server.ini

[root@q conf]# vim server.ini

[system]

brokerId=0

numPartitions=1

serverPort=8123

dashboardHttpPort=8120

unflushThreshold=0

unflushInterval=10000

maxSegmentSize=1073741824

maxTransferSize=1048576

deletePolicy=delete,168

deleteWhen=0 0 6,18 \* \* ?

flushTxLogAtCommit=1

stat=true

;; Update consumers offsets to current max offsets when consumers offsets are out of range of current broker's messages.

;; It must be false in production.But recommend to be true in development or test.

updateConsumerOffsets=true

[zookeeper]

zk.zkConnect=127.0.0.1:2181,127.0.0.1:2182,127.0.0.1:2183

zk.zkSessionTimeoutMs=30000

zk.zkConnectionTimeoutMs=30000

zk.zkSyncTimeMs=5000

;; Topics section

[topic=test]

[topic=meta-test]

[topic=xl] #自定义的消息xl的队列

启动

[root@q conf]# cd ..

[root@q metamorphosis-server-wrapper]# ls

bin conf lib logs provided

[root@q metamorphosis-server-wrapper]# cd bin/

[root@q bin]# sh metaServer.sh start &

查看状态

[root@q bin]# sh metaServer.sh stats

测试

[root@q bin]# ./zkCli.sh -server 127.0.0.1:2181 [zk: 127.0.0.1:2181(CONNECTING) 0] ls /

2017-01-01 19:19:42,232 [myid:] - INFO [main-SendThread(127.0.0.1:2181):ClientCnxn$SendThread@1235] - Session establishment complete on server 127.0.0.1/127.0.0.1:2181, sessionid = 0x159591602330000, negotiated timeout = 30000

WATCHER::

WatchedEvent state:SyncConnected type:None path:null

[zookeeper, meta] #在此目录生成meta

[zk: 127.0.0.1:2181(CONNECTED) 1] ls /meta

[brokers]

[zk: 127.0.0.1:2181(CONNECTED) 2] ls /meta/brokers

[topics-sub, ids, topics-pub, topics]

[zk: 127.0.0.1:2181(CONNECTED) 3] ls /meta/brokers/topics

[meta-test, test, xl]

[zk: 127.0.0.1:2181(CONNECTED) 4] ls /meta/brokers/ids #brok节点

[0]

[zk: 127.0.0.1:2181(CONNECTED) 5] get /meta/brokers/ids/0

meta://192.168.179.161:8123

cZxid = 0x200000046

ctime = Sun Jan 01 16:44:09 CST 2017

mZxid = 0x200000048

mtime = Sun Jan 01 16:44:09 CST 2017

pZxid = 0x200000049

cversion = 2

dataVersion = 1

aclVersion = 0

ephemeralOwner = 0x0

dataLength = 27

numChildren = 2

[zk: 127.0.0.1:2181(CONNECTED) 6]

部署另外一台

在同一台模拟部署

[root@q tools]# cp -rf taobao/ taobao1

[root@q bin]# pwd

/home/zsq/tools/taobao1/metamorphosis-server-wrapper/bin

[root@q bin]# ls

env.bat env.sh localMetaServer.bat log4j.properties metaServer.bat metaServer.sh tools\_log4j.properties

修改脚本metaServer.sh的pid文件

PID\_FILE="$PID\_DIR/.run1.pid"

修改env.sh，默认是9123端口

export JMX\_PORT=9124

启动

[root@q bin]# sh metaServer.sh start $

再次查看

[zk: 127.0.0.1:2181(CONNECTED) 12] ls /meta/brokers/ids #有两个broker就表面集群部署成功

[0, 1]

[zk: 127.0.0.1:2181(CONNECTED) 18] get /meta/brokers/ids/1

meta://192.168.179.161:8124

cZxid = 0x200000092

ctime = Sun Jan 01 21:05:26 CST 2017

mZxid = 0x200000094

mtime = Sun Jan 01 21:05:26 CST 2017

pZxid = 0x200000095

cversion = 2

dataVersion = 1

aclVersion = 0

ephemeralOwner = 0x0

dataLength = 27

numChildren = 2

[zk: 127.0.0.1:2181(CONNECTED) 19]

# 消息中间件metaq的配置启动及相关维护操作



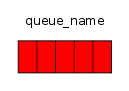
# RabbitMQ简介

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**](http://lib.csdn.net/base/11)、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。  
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。

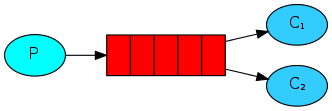
ConnectionFactory、Connection、Channel

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。  
[](http://ostest.qiniudn.com/wordpress/wp-content/uploads/2014/02/queue.png)

RabbitMQ中的消息都只能存储在Queue中，生产者（下图中的P）生产消息并最终投递到Queue中，消费者（下图中的C）可以从Queue中获取消息并消费。  
[qq](http://ostest.qiniudn.com/wordpress/wp-content/uploads/2014/02/qq.png)

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。  
[](http://ostest.qiniudn.com/wordpress/wp-content/uploads/2014/02/2014-2-21-9-46-43.png)

Message acknowledgment

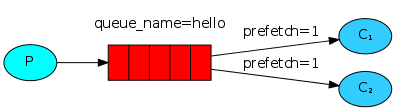
在实际应用中，可能会发生消费者收到Queue中的消息，但没有处理完成就宕机（或出现其他意外）的情况，这种情况下就可能会导致消息丢失。为了避免这种情况发生，我们可以要求消费者在消费完消息后发送一个回执给RabbitMQ，RabbitMQ收到消息回执（Message acknowledgment）后才将该消息从Queue中移除；如果RabbitMQ没有收到回执并检测到消费者的RabbitMQ连接断开，则RabbitMQ会将该消息发送给其他消费者（如果存在多个消费者）进行处理。这里不存在timeout概念，一个消费者处理消息时间再长也不会导致该消息被发送给其他消费者，除非它的RabbitMQ连接断开。  
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。

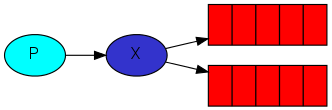
Message durability

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。  
[](http://ostest.qiniudn.com/wordpress/wp-content/uploads/2014/02/2014-2-21-9-49-08.png)

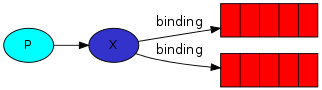
Exchange

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）。  
[](http://ostest.qiniudn.com/wordpress/wp-content/uploads/2014/02/2014-2-21-9-51-03.png)  
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。  
[](http://ostest.qiniudn.com/wordpress/wp-content/uploads/2014/02/2014-2-21-9-52-46.png)

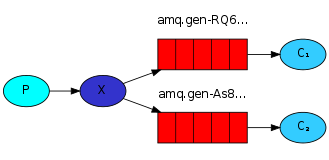
Binding key

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。

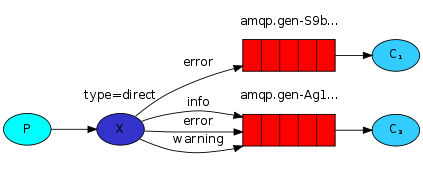
Exchange Types

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。  
[](http://ostest.qiniudn.com/wordpress/wp-content/uploads/2014/02/2014-2-21-9-54-26.png)  
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。  
[](http://ostest.qiniudn.com/wordpress/wp-content/uploads/2014/02/2014-2-21-9-55-20.png)  
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。

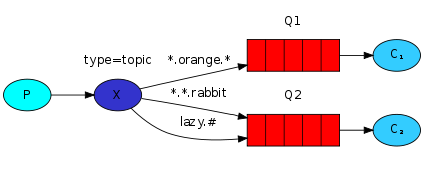
topic

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

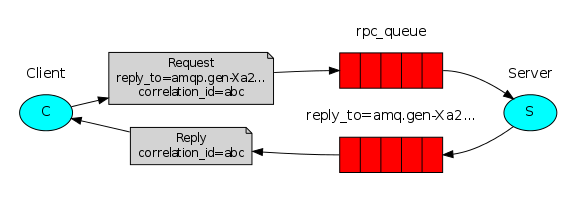
binding key中可以存在两种特殊字符“\*”与“#”，用于做模糊匹配，其中“\*”用于匹配一个单词，“#”用于匹配多个单词（可以是零个）

[](http://ostest.qiniudn.com/wordpress/wp-content/uploads/2014/02/2014-2-21-9-57-37.png)  
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。

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。  
[](http://ostest.qiniudn.com/wordpress/wp-content/uploads/2014/02/2014-2-21-9-59-04.png)  
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

总结

本文介绍了RabbitMQ中个人认为最重要的概念，充分利用RabbitMQ提供的这些功能就可以处理我们绝大部分的异步业务了。

转自<http://blog.csdn.net/whycold/article/details/41119807#t18>

# RabbitMQ 中文文档

<http://rabbitmq.mr-ping.com/>

# 消息中间件rabbitmq的安装部署

[root@q tools]#mkdir rabbitmq && cd rabbitmq/ #创建并进入目录

[root@q rabbitmq]# ls

epel-release-6-8.noarch.rpm

[root@q rabbitmq]# rpm -ivh epel-release-6-8.noarch.rpm #安装epel源

[root@q rabbitmq]# yum install rabbitmq-server -y

查看插件

[root@q ~]# /usr/lib/rabbitmq/bin/rabbitmq-plugins list

[ ] amqp\_client 3.1.5

[ ] cowboy 0.5.0-rmq3.1.5-git4b93c2d

[ ] eldap 3.1.5-gite309de4

[ ] mochiweb 2.7.0-rmq3.1.5-git680dba8

[ ] rabbitmq\_amqp1\_0 3.1.5

[ ] rabbitmq\_auth\_backend\_ldap 3.1.5

[ ] rabbitmq\_auth\_mechanism\_ssl 3.1.5

[ ] rabbitmq\_consistent\_hash\_exchange 3.1.5

[ ] rabbitmq\_federation 3.1.5

[ ] rabbitmq\_federation\_management 3.1.5

[ ] rabbitmq\_jsonrpc 3.1.5

[ ] rabbitmq\_jsonrpc\_channel 3.1.5 #主要看这个 ，下面是一些监控

[ ] rabbitmq\_jsonrpc\_channel\_examples 3.1.5

[ ] rabbitmq\_management 3.1.5

[ ] rabbitmq\_management\_agent 3.1.5

[ ] rabbitmq\_management\_visualiser 3.1.5

[ ] rabbitmq\_mqtt 3.1.5

[ ] rabbitmq\_shovel 3.1.5

[ ] rabbitmq\_shovel\_management 3.1.5

[ ] rabbitmq\_stomp 3.1.5

[ ] rabbitmq\_tracing 3.1.5

[ ] rabbitmq\_web\_dispatch 3.1.5

[ ] rabbitmq\_web\_stomp 3.1.5

[ ] rabbitmq\_web\_stomp\_examples 3.1.5

[ ] rfc4627\_jsonrpc 3.1.5-git5e67120

[ ] sockjs 0.3.4-rmq3.1.5-git3132eb9

[ ] webmachine 1.10.3-rmq3.1.5-gite9359c7

[root@q rabbitmq]# /usr/lib/rabbitmq/bin/rabbitmq-plugins enable rabbitmq\_management #开启管理界面

The following plugins have been enabled:

mochiweb

webmachine

rabbitmq\_web\_dispatch

amqp\_client

rabbitmq\_management\_agent

rabbitmq\_management

Plugin configuration has changed. Restart RabbitMQ for changes to take effect.

[root@q rabbitmq]# /etc/init.d/rabbitmq-server start

Starting rabbitmq-server: SUCCESS

rabbitmq-server.

[root@q rabbitmq]# rabbitmqctl list\_queues #出现如下这个表示安装成功

Listing queues ...

...done.

[root@q rabbitmq]# rabbitmqctl add\_vhost zsq #创建Vhost

Creating vhost "zsq" ...

...done.

[root@q rabbitmq]# rabbitmqctl list\_vhosts #遍历所有虚拟主机信息

Listing vhosts ...

/ #这个是默认的

zsq

...done.

[root@q rabbitmq]# rabbitmqctl add\_user zsq 13456 #创建用户跟密码

Creating user "zsq" ...

...done.

[root@q rabbitmq]# rabbitmqctl list\_users #查看用户信息

Listing users ...

guest [administrator] #默认

zsq [] #自己创建的

...done.

[root@q rabbitmq]# rabbitmqctl set\_permissions -p zsq zsq ".\*" ".\*" ".\*" #vhost跟用户绑定

Setting permissions for user "zsq" in vhost "zsq" ...

...done.

其它命令

rabbitmqctl list\_queues #显示队列

rabbitmqctl delete\_vhost vhostname #删除Vhost

其它的由开发程序来实现，创建的vhost 用户 密码 IP等信息给开发

rabbitmq配置

一般情况下，RabbitMQ的默认配置就足够了。如果希望特殊设置的话，有两个途径：

一个是环境变量的配置文件 /etc/rabbitmq/rabbitmq-env.conf ；

一个是配置信息的配置文件 /etc/rabbitmq/rabbitmq.conf；

注意，这两个文件默认是没有的，如果需要必须自己创建。

管理web界面登陆

在浏览器中输入http://127.0.0.1:15672/

输入用户名和密码（默认为guest）

