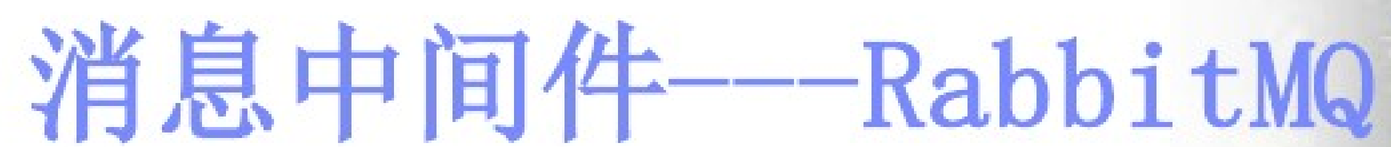


谢茂盛

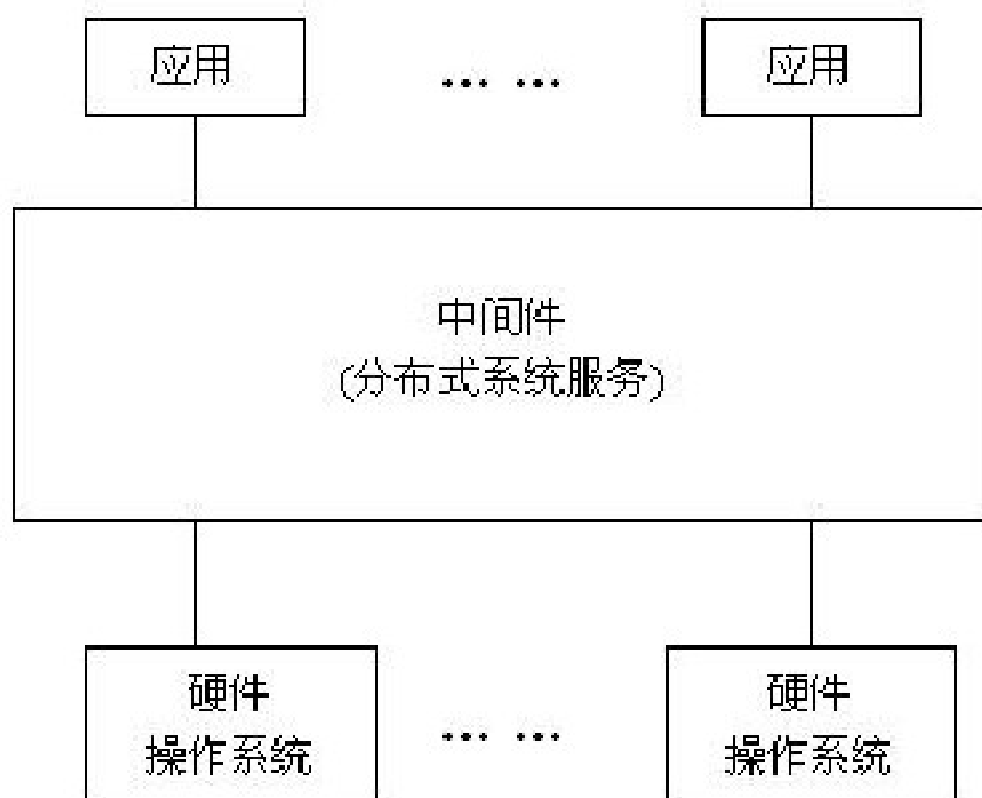
© Copyright ND Corporation 2015

中间件的简介

➢中间件的定义

中间件的定义为：中间件是一种独立的系统软件或服务程序，分布式应用软件借助这种软件在不同的技术之间共享资源，中间件定位于客户机服务器的操作系统之上，管理计算机资源和网络通信。

因而中间件是指一类软件，是基于分布式处理的软件，最突出的特点是其网络通信功能。也可认为中间件是位于平台和应用之间的通用服务，这些服务具有标准的程序接口和协议。针对不同的操作系统和硬件平台，可以有符合接口和协议的多种实现。



中间件的分类

中间件可分为六类：

1) 终端仿真/屏幕转换

2) 数据访问中间件（UDA）

3) 远程过程调用中间件（RPC）

4) 消息中间件（MOM）

5) 交易中间件（TPM）

6) 对象中间件

消息中间件

➢什么是消息中间件

面向消息的中间件（MOM），提供了以松散耦合的灵活方式集成应用程序的一种机制。它们提供了基于存储和转发的应用程序之间的异步数据发送，即应用程序彼此不直接通信，而是与作为中介的MOM通信。MOM提供了有保证的消息发送（至少是在尽可能地做到这一点），应用程序开发人员无需了解远程过程调用（PRC）和网络/通信协议的细节。

➢消息中间件简介

消息中间件利用高效可靠的消息传递机制进行平台无关的数据交流，并基于数据通信来进行分布式系统的集成。通过提供消息传递和消息排队模型，它可以在分布式环境下扩展进程间的通信。

消息中间件适用于需要可靠的数据传送的分布式环境。采用消息中间件机制的系统中，不同的对象之间通过传递消息来激活对方的事件，完成相应的操作。发送者将消息发送给消息服务器，消息服务器将消息存放在若干队列中，在合适的时候再将消息转发给接收者。消息中间件能在不同平台之间通信，它常被用来屏蔽掉各种平台及协议之间的特性，实现应用程序之间的协同，其优点在于能够在客户和服务器之间提供同步和异步的连接，并且在任何时刻都可以将消息进行传送或者存储转发，这也是它比远程过程调用更进一步的原因。

MOM将消息路由给应用程B，这样消息就可以存在于完全不同的计算机上，MOM 负责处理网络通信。如果网络连接不可用，MOM会存储消息，直到连接变得可用时，再将消息转发给应用程序B。

灵活性的另一方面体现在，当应用程序A发送其消息时，应用程序B甚至可以不处于执行状态。MOM将保留这个消息，直到应用程序B开始执行并试着检索消息为止。这还防止了应用程序A因为等待应用程序B检索消息而出现阻塞。这种异步通信要求应用程序的设计与现在大多数应用程序不同，不过，对于时间无关或并行处理，它可能是一个极其有用的方法。

消息中间件与分布式对象调用的比较

➢分布式对象调用

如CORBA，RMI和DCOM，提供了一种通讯机制，透明地在异构的分布式计算环境中传递对象请求，而这些对象可以位于本地或远程机器。它通过在对象与对象之间提供一种统一的接口，使对象之间的调用和数据共享不再关心对象的位置、实现语言及所驻留的操作系统。这个接口就是面向对象的中间件。

尽管面向对象的中间件是一种很强大的规范被广泛应用，但是面对大规模的复杂分布式系统，这些技术也显示出了局限性：

1．同步通信：客户发出调用后，必须等待服务对象完成处理并返回结果后才能继续执行。

2．客户和服务对象的生命周期紧密耦合：客户进程和服务对象进程都必须正常运行，如果由于服务对象崩溃或网络故障导致客户的请求不可达，客户会接收到异常

➢消息中间件

为了解决这些问题，出现了面向消息的中间件，它较好地解决了以上的问题。

消息中间件作为一个中间层软件，它为分布式系统中创建、发送、接收消息提供了一套可靠通用的方法，实现了分布式系统中可靠的、高效的、实时的跨平台数据传输。消息中间件减少了开发跨平台和网络协议软件的复杂性，它屏蔽了不同操作系统和网络协议的具体细节，面对规模和复杂度都越来越高的分布式系统，消息中间件技术显示出了它的优越性：

1．采用异步通信模式：发送消息者可以在发送消息后进行其它的工作，不用等待接收者的回应，而接收者也不必在接到消息后立即对发送者的请求进行处理；

2．客户和服务对象生命周期的松耦合关系：客户进程和服务对象进程不要求都正常运行，如果由于服务对象崩溃或者网络故障导致客户的请求不可达，客户不会接收到异常，消息中间件能保证消息不会丢失。

消息中间件的传递模式

消息中间件一般有两种传递模型：点对点模型（PTP）和发布-订阅模型（Pub/Sub）

➢点对点模型（PTP）

点对点模型用于消息生产者和消息消费者之间点到点的通信。消息生产者将消息发动到由某个名字标识的特定消费者。这个名字实际上对应于消息服务中的一个队列（Queue），在消息传动给消费者之前它被存储在这个队列中。队列可以是持久的，以保证在消息服务出现故障时仍然能够传递消息。

➢发布-订阅模型（Pub/Sub）

发布-订阅模型用称为主题（topic）的内容分层结构代替了PTP模型中的惟一目的地，发送应用程序发布自己的消息，指出消息描述的是有关分层结构中的一个主题的信息。希望接收这些消息的应用程序订阅了这个主题。订阅包含子主题的分层结构中的主题的订阅者可以接收该主题和其子主题发表的所有消息。

发布-订阅消息模式

➢订阅杂志

我们很多人都订过杂志，其过程很简单。只要告诉邮局我们所要订的杂志名、投递的地址，付了钱就OK。出版社定期会将出版的杂志交给邮局，邮局会根据订阅的列表，将杂志送达消费者手中。这样我们就可以看到每一期精彩的杂志了。

几个特点

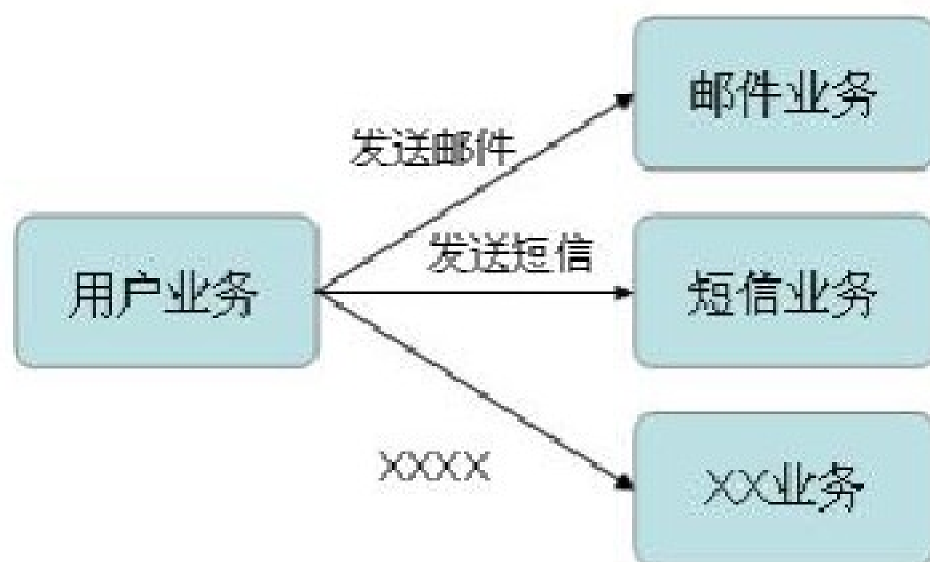
- 消费者订杂志不需要直接找出版社
- 出版社只需要把杂志交给邮局
- 邮局将杂志送达消费者

邮局在整个过程中扮演了非常重要的中转作用，在出版社和消费者相互不需要知道对方的情况下，邮局完成了杂志的投递。

➢发布-订阅消息模式

刚刚讲了订阅杂志，下面我们会讲传统调用模式演化到发布-订阅消息模式。

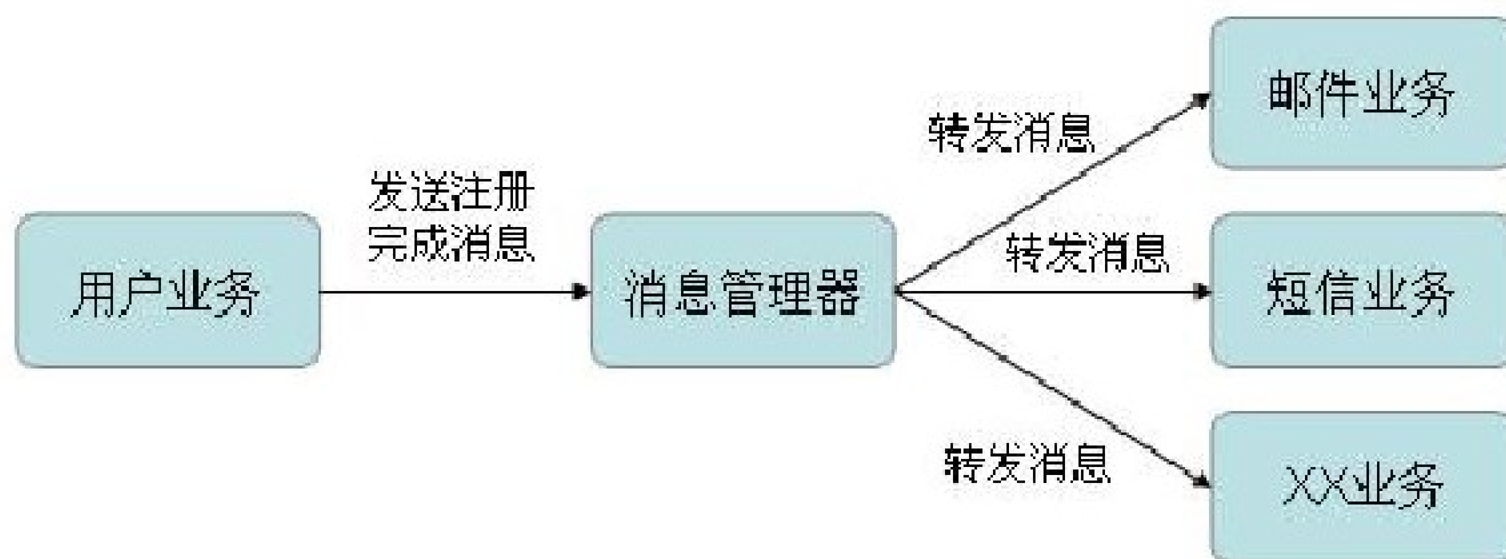
有些网站在注册用户成功后发一封激活邮件，用户收到邮件后点击激活链接后才能使用该网站。一般的做法是在注册用户业务逻辑中调用发送邮件的逻辑。这 样用户业务就依赖于邮件业务。如果以后改为短信激活，注册用户业务逻辑就必须修改为调用发送短信的逻辑。如果要注册后给用户加点积分，再加一段逻辑。经过 多次修改，我们发现很简单的注册用户业务已经越来越复杂，越来越难以维护。相信很多开发者都会有类似痛苦的经历。



即使用户业务实现中对其他业务是接口依赖，也避免不了业务变化带来的依赖影响。怎么办？解耦！将注册用户业务逻辑中注册成功后的处理剥离出来。

再回头看看"订阅杂志"，如果没有邮局，出版社就必须自己将杂志送达所有消费者。这种情形就和现在的注册用户业务一样。我们发现问题了，在用户业务和其他业务之间缺少了邮局所扮角色。

我们把邮局抽象成一个管理消息的地方，叫"消息管理器"。注册用户成功后发送一个消息给消息管理器，由消息管理器转发该消息给需要处理的业务。现在，用户业务只依赖于消息管理器了，它再也不会为了注册用户成功后的其他处理而烦恼。



注册用户的改造就是借鉴了"订阅杂志"这样原始的模式。我们再进一步抽象，用户业务就是消息的"生产者"，它将消息发布到消息管理器。邮件业务就是 消息的"消费者"，它将收到的消息进行处理。邮局可以订阅很多种杂志，杂志都是通过某种编号来区分；消息管理器也可以管理多种消息，每种消息都会有一个 "主题"来区分，消费者都是通过主题来订阅的。

消息队列（MQ）相关概念

- **消息（message）**

　　消息是MQ中最小的概念，本质上就是一段数据，它能被一个或者多个应用程序所理解，是应用程序之间传递的信息载体。

- **队列（Queue）**

　　负责存储消息。

- **队列管理器(Queue Manager)**

　　队列管理器是一个负责向应用程序提供消息服务的机构，如果把队列管理器比作数据库，那么队列就是其中一张表。

- **通道(Channel)**

　　通道是两个管理器之间的一种单向点对点的的通信连接，如果需要双向交流，可以建立一对通道。

- **5.监听器(listner)**

　　监听消息的接收

消息队列（MQ）的特性

➢可靠性传输

这个特点可以说是消息中间件的立足之本，对于应用来说，只要成功把数据提交给消息中间件，那么关于数据可靠传输的问题就由消息中间件来负责。

➢不重复传输

不重复传播也就是断点续传的功能，特别适合网络不稳定的环境，节约网络资源。

➢异步性传输

异步性传输是指，接受信息双方不必同时在线，具有脱机能力和安全性。

➢消息驱动

接到消息后主动通知消息接收方。

➢支持事务

应用程序可以把一些数据更新组合成一个工作单元，这些更新通常是逻辑相关的，为了保障数据完整性，所有的更新必须同时成功或者同时失败）。

常用MQ产品比较

	ActiveMQ	Joram	HornetQ	OpenMQ	MuleMQ	SonicMQ	RabbitMQ	ZeroMQ
关注度	高	中	中	中	低	低	高	中
成熟度	成熟	比较成熟	比较成熟	比较成熟	新产品无成功案例	成熟	成熟	不成熟
所属社区/公司	Apache	OW2	Jboss	Sun	Mule	Progress		
社区活跃度	高	中	中	低	高	低	高	低
文档	多	多	中	中	少	少	多	中
特点	功能齐全，被大量开源项目使用		在Linux平台上直接调用操作系统的AIO，性能得到很大的提升		性能非常好，与MuleESB无缝整合	性能优越的商业MQ	由于Erlang语言的并发能力，性能很好	低延时，高性能，最高43万条消息每秒
授权方式	开源	开源	开源	开源	商业	商业	开源	开源
开发语言	Java	Java	Java	Java	Java	Java	Erlang	C
支持的协议	OpenWire、STOMP、REST、XMPP、AMQP	JMS	JMS	JMS	JMS	JMS	AMQP	TCP、UDP
客户端支持语言	Java、C、C++、Python、PHP、Perl、.net等	Java	Java	Java	Java	Java、C、C++、.net	Java、C、C++、Python、PHP、Perl等	python、java、php、.net等
持久化	内存、文件、数据库	内存、文件	内存、文件	内存、文件	内存、文件	内存、文件、数据库	内存、文件	在消息发送端保存
事务	支持	支持	支持	支持	支持	支持	不支持	不支持
集群	支持	支持	支持	支持	支持	支持	支持	不支持
负载均衡	支持	支持	支持	支持	支持	支持	支持	不支持
管理界面	一般	一般	无	一般	一般	好	无	无
部署方式	独立、嵌入	独立、嵌入	独立、嵌入	独立、嵌入	独立	独立	独立	独立
评价	成熟稳定，开源首选	依赖容器，不适合跨语言调用	推出的时间不长，尚无使用案例，不适合跨语言调用	依赖容器，不适合跨语言调用	推出的时间不长，无成功案例，目前仅支持Java	成熟稳定	Queue的数量大于50后，高并发下无法持续稳定的提供服务	不支持事务、集群，并且消息不能在服务端持久化

MQ适用场景介绍

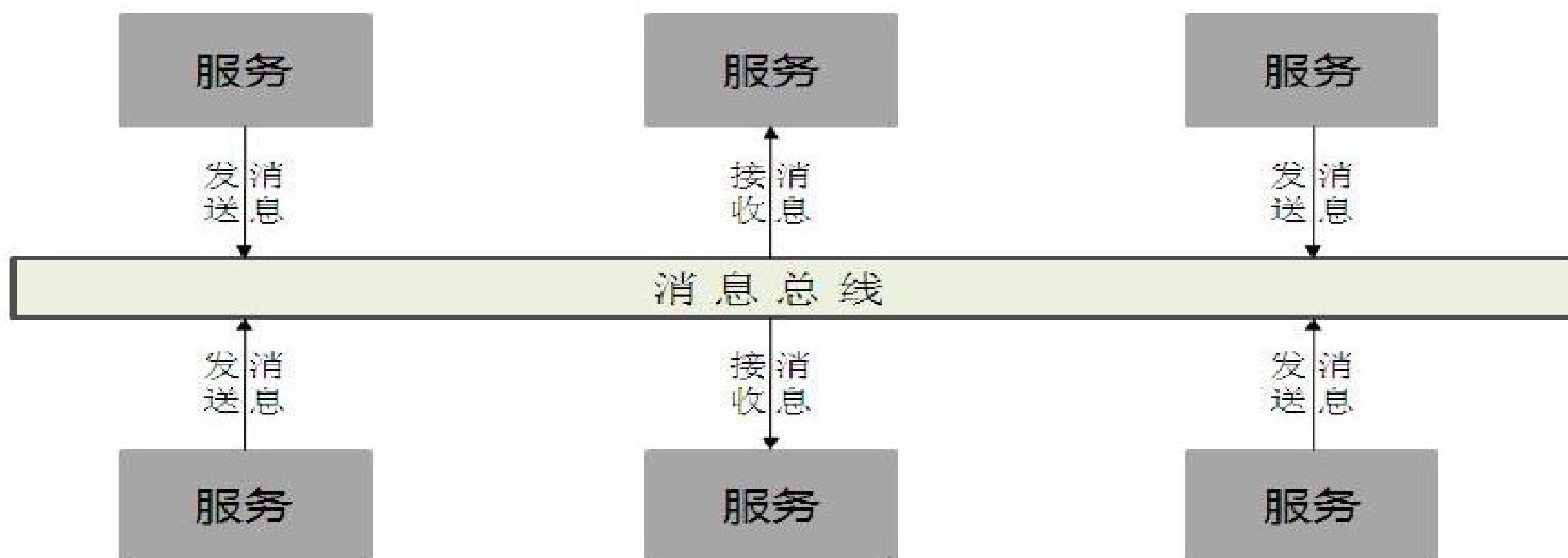
MQ消息队列是应运松偶合的概念而产生的，主要以队列和发布订阅为消息传输机制，以异步的方式将消息可靠的传输到消费端的一种基础产品。它被广泛的应用与跨平台、跨系统的分布式系统之间，为它们提供高效可靠的异步传输机制。

➢消息通道（Message Channel）

如客户端与服务端需要安全可靠的交互，可以将一个MQ的队列作为安全通道，是客户端与服务端能够安全高效的进行异步通讯。

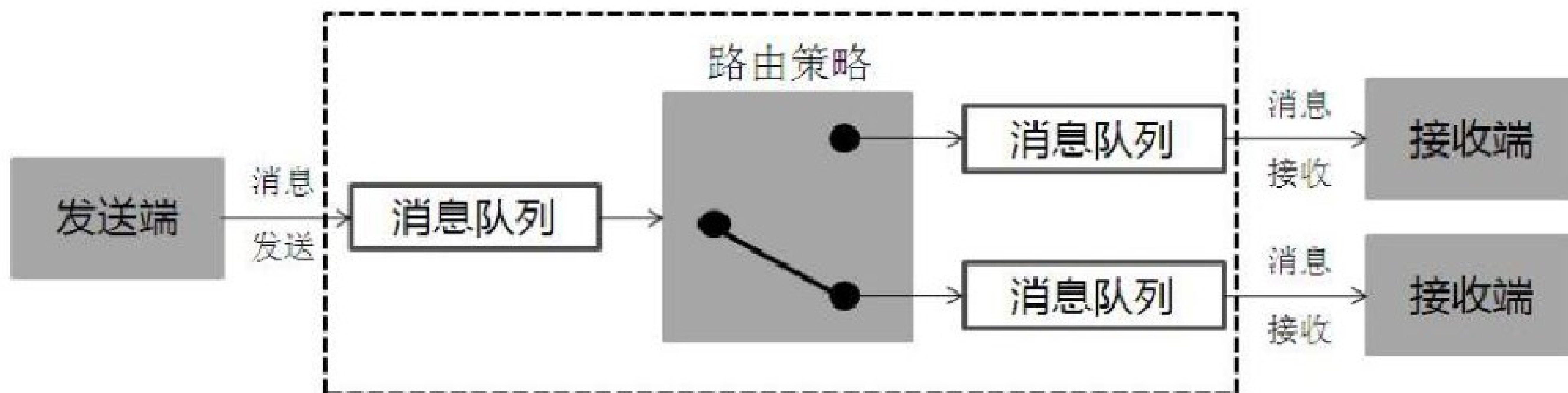
➢消息总线（Message Bus）

对于由许多独立开发的服务组成的分布式系统，倘若要将它们组成一个完整的系统，这些服务必须能够可靠地交互，同时，为了系统的健壮性，每个服务之间又不能产生过分紧密的依赖关系，这样就可以通过消息总线将不同的服务连接起来，允许它们异步的传递数据。



➢消息路由（Message Router）

通过消息路由，可以将发送到MQ指定队列的消息根据规则路由到不同的队列。



RabbitMQ 简介与使用

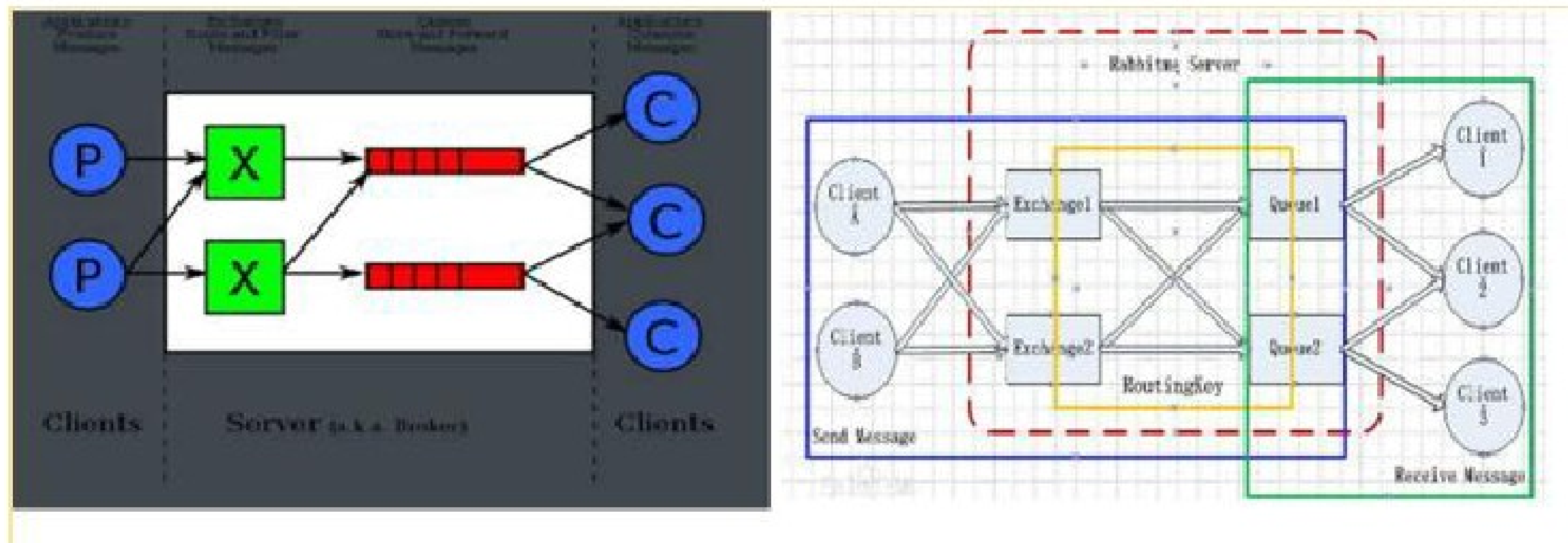
➢RabbitMQ

RabbitMQ是一种处理消息验证、消息转换和消息路由的架构模式，它协调应用程序之间的信息通信，并使得应用程序或者软件模块之间的相互意识最小化，有效实现解耦。

RabbitMQ适合部署在一个拓扑灵活易扩展的规模化系统环境中，有效保证不同模块、不同节点、不同进程之间消息通信的时效性；而且，RabbitMQ特有的集群HA安全保障能力可以实现信息枢纽中心的系统级备份，同时单节点具备消息恢复能力，当系统进程崩溃或者节点宕机时，RabbitMQ正在处理的消息队列不会丢失，待节点重启之后可根据消息队列的状态数据以及信息数据及时恢复通信。

RabbitMQ在功能性、时效性、安全可靠性以及SLA方面的出色能力可有效支持OpenStack云平台系统的规模化部署、弹性扩展、灵活架构以及信息安全的需求。

➢ RabbitMQ结构图



➢ RabbitMQ一些概念

Brocker：消息队列服务器实体。

Exchange：消息交换机，指定消息按什么规则，路由到哪个队列。

Queue：消息队列，每个消息都会被投入到一个或者多个队列里。

Binding：绑定，它的作用是把exchange和queue按照路由规则binding起来。

Routing Key：路由关键字，exchange根据这个关键字进行消息投递。

Vhost：虚拟主机，一个broker里可以开设多个vhost，用作不用用户的权限分离。

Producer：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。

Channel：消息通道，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。

➢ 消息队列的使用过程

1.客户端连接到消息队列服务器，开一个channel。

2.客户端声明一个exchange，并设置相关的属性。

3.客户端声明一个queue，并设置相关属性

4.客户端使用routing key，在exchange和queue之间建立好绑定关系

5.客户端投递消息到exchange。

6.exchange接收到消息后，就根据消息的key和已经设置的binding，进行消息的路由，将消息投递到一个或多个队列里。

7.exchange也有几个类型，完全根据key进行投递的叫做Direct交换机，例如，绑定时设置了routing key为 “abc” ，那么客户端提交的消息，只有设置了key为"abc” 的才会投递到队列。对key进行模式匹配后进行投递的叫做Topic交换机，符号”#” 匹配一个或者多个词，符号 “*” 匹配正好一个词。例如 “abc.#” 匹配 “abc.def.ghi”， “abc.*” 只匹配 “abc.def” 。还有一种不要key的，叫做Fanout交换机，它采用广播模式，一个消息进来时投递到与该交换机绑定的所有队列.

RabbitMQ 原理---AMQP(高级消息队列协议)的实现

➢AMQP

AMQP是应用层协议的一个开放标准，为面向消息的中间件而设计，其中RabbitMQ是AMQP协议的一个开源实现，OpenStack Nova各软件模块通过AMQP协议实现信息通信。AMQP协议的设计理念与数据通信网络中的路由协议非常类似，可归纳为基于状态的面向无连接通信系统模式。不同的是，数据通信网络是基于通信链路的状态决定客户端与服务端之间的链接，而AMQP是基于消息队列的状态决定消息生产者与消息消费者之间的链接。对于AMQP来讲，消息队列的状态信息决定通信系统的转发路径，链接两端之间的链路并不是专用且永久的，而是根据消息队列的状态与属性实现信息在RabbitMQ服务器上的存储与转发，正如数据通信网络的IP数据包转发机制，所有的路由器是基于通信链路的状态而形成路由表，IP数据包根据路由表实现报文的本地存储与逐级转发，二者在实现机制上具有异曲同工之妙。

➢ AMQP三要素

AMQP有两个核心要素——交换器（Exchange）与队列（Queue）通过消息的绑定与转发机制实现信息通信。其中，交换器是由消费者应用程序创建，并且可与其他应用程序实现共享服务，其功能与数据通信网络中的路由器非常相似，即接收消息之后通过路由表将消息准确且安全的转发至相应的消息队列。一台RabbitMQ服务器或者由多台RabbitMQ服务器组成的集群可以存在多个交换器，每个交换器通过唯一的Exchange ID进行识别。

交换器根据不同的应用程序的需求，在生命周期方面也是灵活可变的，主要分为三种：持久交换器、临时交换器与自动删除交换器。持久交换器是在RabbitMQ服务器中长久存在的，并不会因为系统重启或者应用程序终止而消除，其相关数据长期驻留在硬盘之上；临时交换器驻留在内存中，随着系统的关闭而消失；自动删除交换器随着宿主应用程序的中止而自动消亡，可有效释放服务器资源。

队列也是由消费者应用程序创建，主要用于实现存储与转发交换器发送来的消息，队列同时也具备灵活的生命周期属性配置，可实现队列的持久保存、临时驻留与自动删除。

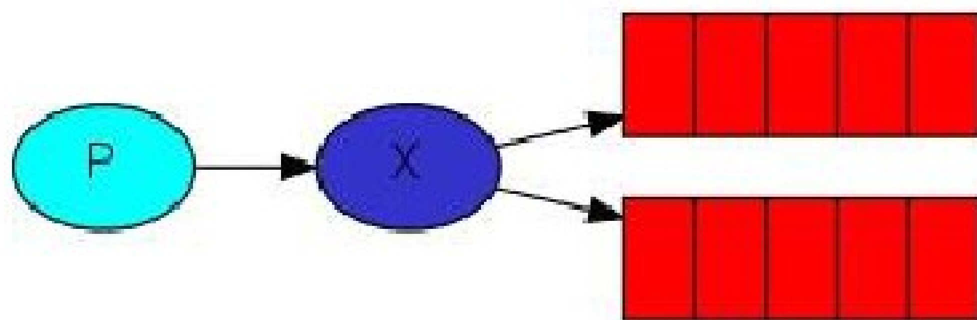
由以上可以看出，消息、队列和交换器是构成AMQP的三个关键组件，任何一个组件的实效都会导致信息通信的中断，因此鉴于三个关键组件的重要性，系统在创建三个组件的同时会打上“Durable”标签，表明在系统重启之后立即恢复业务功能。

➢ AMQP三种类型交换机

1 广播式交换器类型（fanout）

该类交换器不分析所接收到消息中的Routing Key，默认将消息转发到所有与该交换器绑定的队列中去。广播式交换器转发效率最高，但是安全性较低，消费者应用程序可获取本不属于自己的消息。

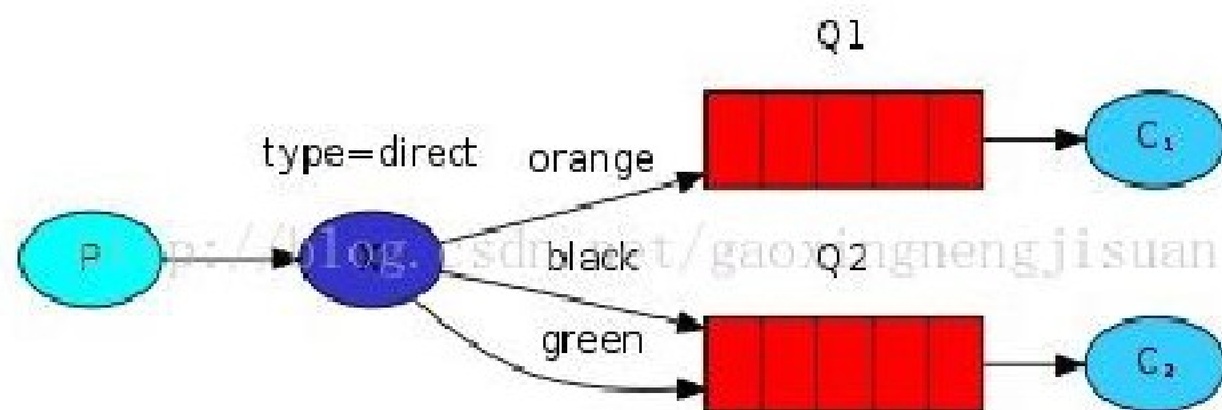
广播交换器是最简单的一种类型，就像我们从字面上理解到的一样，它把所有接受到的消息广播到所有它所知道的队列中去，不论消息的关键字是什么，消息都会被路由到和该交换器绑定的队列中去。



2 直接式交换器类型（direct）

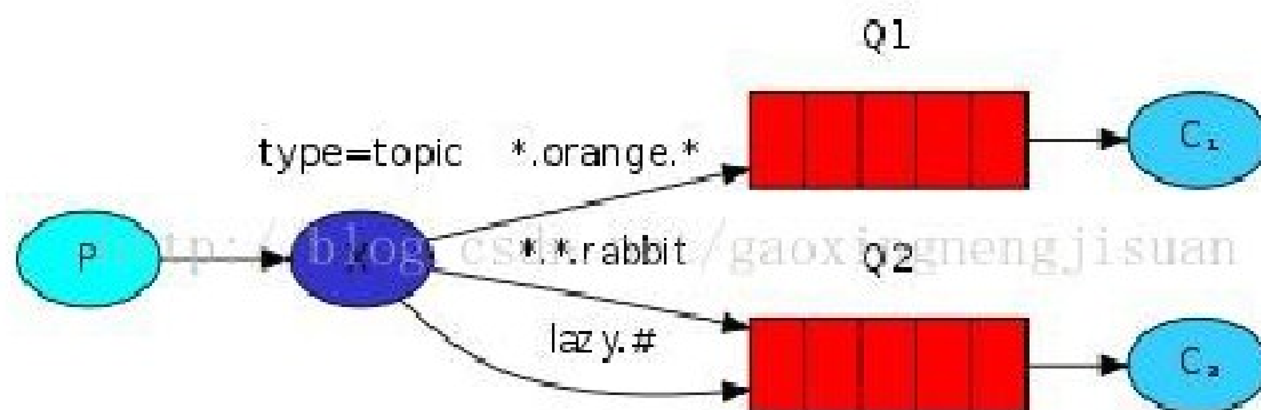
该类交换器需要精确匹配Routing Key与BindingKey，如消息的Routing Key = Cloud，那么该条消息只能被转发至Binding Key = Cloud的消息队列中去。直接式交换器的转发效率较高，安全性较好，但是缺乏灵活性，系统配置量较大。

相对广播交换器来说，直接交换器可以给我们带来更多的灵活性。直接交换器的路由算法很简单——一个消息的routing_key完全匹配一个队列的 binding_key，就将这个消息路由到该队列。绑定的关键字将队列和交换器绑定到一起。当消息的routing_key和多个绑定关键字匹配时消息 可能会被发送到多个队列中。



3 主题式交换器（Topic Exchange）

该类交换器通过消息的Routing Key与Binding Key的模式匹配，将消息转发至所有符合绑定规则的队列中。Binding Key支持通配符，其中"*"匹配一个词组，"#"匹配多个词组（包括零个）。例如，Binding Key="*.Cloud.#"可转发Routing Key="OpenStack.Cloud.GD.GZ"、"OpenStack.Cloud.Beijing"以及"OpenStack.Cloud"的消息，但是对于Routing Key="Cloud.GZ"的消息是无法匹配的。



Spring RabbitMQ HelloWorld 实例

➢ Spring Rabbitmq 配置实例

1.首先是生产者配置

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:rabbit="http://www.springframework.org/schema/rabbit"
       xsi:schemaLocation="
            http://www.springframework.org/schema/beans
                http://www.springframework.org/schema/beans/spring-beans.xsd
            http://www.springframework.org/schema/context
                http://www.springframework.org/schema/context/spring-context.xsd
            http://www.springframework.org/schema/rabbit
                http://www.springframework.org/schema/rabbit/spring-rabbit-1.0.xsd">


    <!-- 连接服务配置  -->
    <rabbit:connection-factory id="connectionFactory" host="localhost" username="guest"
                               password="guest" port="5672"  />

    <rabbit:admin connection-factory="connectionFactory"/>

    <!-- queue 队列声明-->
    <rabbit:queue id="queue_one" durable="true" auto-delete="false" exclusive="false" name="queue_one"/>


    <!-- exchange queue binging key 绑定 -->
    <rabbit:direct-exchange name="my-mq-exchange" durable="true" auto-delete="false" id="my-mq-exchange">
        <rabbit:bindings>
            <rabbit:binding queue="queue_one" key="queue_one_key"/>
        </rabbit:bindings>
    </rabbit:direct-exchange>

    <-- spring amqp默认的是jackson 的一个插件,目的将生产者生产的数据转换为json存入消息队列, 由于fastjson的速度快于jackson,这里替换为fastjson的一个实现 -->
    <bean id="jsonMessageConverter"  class="mq.convert.FastJsonMessageConverter"></bean>

    <-- spring template声明-->
    <rabbit:template exchange="my-mq-exchange" id="amqpTemplate"  connection-factory="connectionFactory"  message-converter="jsonMessageConverter"/>
</beans>
```

2.生产者端调用

```java
import java.util.List;

import org.springframework.amqp.core.AmqpTemplate;


public class MyMqGatway {

        @Autowired
        private AmqpTemplate amqpTemplate;

        public void sendDataToCrQueue(Object obj) {
                amqpTemplate.convertAndSend("queue_one_key", obj);
        }
}
```

3.消费者端配置(与生产者端大同小异)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:rabbit="http://www.springframework.org/schema/rabbit"
       xsi:schemaLocation="
            http://www.springframework.org/schema/beans
                http://www.springframework.org/schema/beans/spring-beans.xsd
            http://www.springframework.org/schema/context
                http://www.springframework.org/schema/context/spring-context.xsd
            http://www.springframework.org/schema/rabbit
                http://www.springframework.org/schema/rabbit/spring-rabbit-1.0.xsd">


    <!-- 连接服务配置  -->
    <rabbit:connection-factory id="connectionFactory" host="localhost" username="guest"
                               password="guest" port="5672"  />

    <rabbit:admin connection-factory="connectionFactory"/>

    <!-- queue 队列声明-->
    <rabbit:queue id="queue_one" durable="true" auto-delete="false" exclusive="false" name="queue_one"/>


    <!-- exchange queue binging key 绑定 -->
    <rabbit:direct-exchange name="my-mq-exchange" durable="true" auto-delete="false" id="my-mq-exchange">
        <rabbit:bindings>
            <rabbit:binding queue="queue_one" key="queue_one_key"/>
        </rabbit:bindings>
    </rabbit:direct-exchange>



    <!-- queue litener  观察 监听模式 当有消息到达时会通知监听在对应的队列上的监听对象-->
    <rabbit:listener-container connection-factory="connectionFactory" acknowledge="auto" task-executor="taskExecutor">
        <rabbit:listener queues="queue_one" ref="queueOneLitener"/>
    </rabbit:listener-container>
</beans>
```

4.消费者端调用

```
import org.springframework.amqp.core.Message;
import org.springframework.amqp.core.MessageListener;

public class QueueOneLitener implements  MessageListener{
        @Override
        public void onMessage(Message message) {
                System.out.println(" data :" + message.getBody());
        }
}
```

Thank You