**MQ架构**

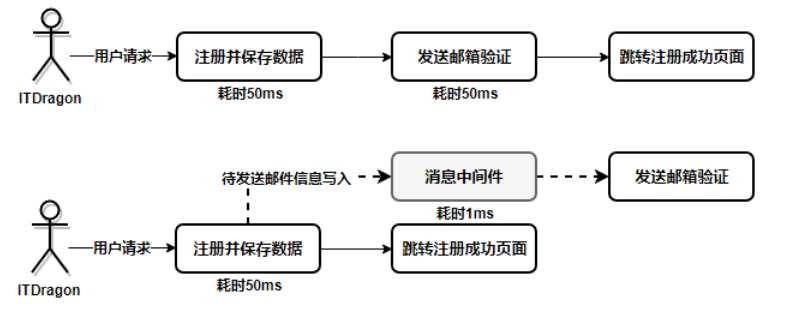
## 一、消息队列

**消息中间件是大型分布式系统中不可或缺的重要组件。它使用简单，却解决了不少难题，比如异步处理，系统藕合，流量削锋，分布式事务管理等。实现了一个高性能，高可用，高扩展的系统。**

**1 处理方式**

**异步处理：**调用者发起请求后，调用者不会立刻得到结果，也无需等待结果，继续执行其他业务逻辑。提高了效率但存在异步请求失败的隐患，适用于非核心业务逻辑处理。

**同步处理：**调用者发起请求后，调用者必须等待直到返回结果，再根据返回的结果执行其他业务逻辑。效率虽然没有异步处理高，但能保证业务逻辑可控性，适用于核心业务逻辑处理。



**2 异步处理**

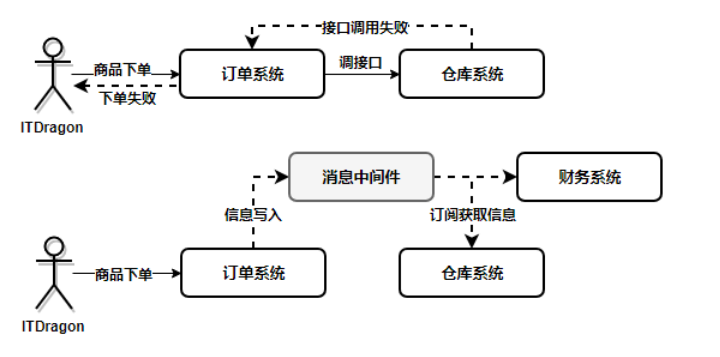
一、在传统的系统架构，用户从注册到跳转成功页面，中间需要等待邮件发送的业务逻辑耗时。这不仅影响系统响应时间，降低了CPU吞吐量，同时还影响了用户的体验。

二、通过消息中间件将邮件发送的业务逻辑异步处理，用户注册成功后发送数据到消息中间件，再跳转成功页面，邮件发送的逻辑再由订阅该消息中间件的其他系统负责处理，

三、消息中间件的读写速度非常的快，其中的耗时可以忽略不计。通过消息中间件可以处理更多的请求。

**3 系统藕合和事务的最终一致性**

分布式系统是若干个独立的计算机（系统）集合。每个计算机负责自己的模块，实现系统的解耦，也避免单点故障对整个系统的影响。每个系统还可以做一个集群，进一步降低故障的发生概率。在这样的分布式系统中，消息中间件又扮演着什么样的角色呢？举一个比较常见的应用场景：订单系统下单成功后，需要调用仓库系统接口，选择最优的发货仓库和更新商品库存。若因为某种原因在调用仓库系统接口失败，会直接影响到下单流程。

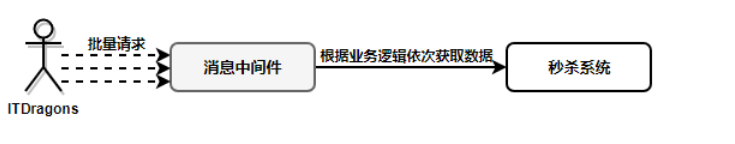


一、消息中间件可以让各系统之间耦合性降低，不会因为其他系统的异常影响到自身业务逻辑。各尽其职，订单系统只需负责将订单数据持久化到数据库中，仓库系统只需负责更新库存，不会因为仓库系统的原因从而影响到下单的流程。

二、各位看官是否发现了一个问题，下单和库存减少本应该是一个事务。因为分布式的原因很难保证事务的强一致性。这里通过消息中间件实现事务的最终一致性效果

**4 流量削锋**

也称限流。在秒杀，抢购的活动中，为了不影响整个系统的正常使用，一般会通过消息中间件做限流，避免流量突增压垮系统，前端页面可以提示"排队等待"，即便用户体验很差，也不能让系统垮掉。



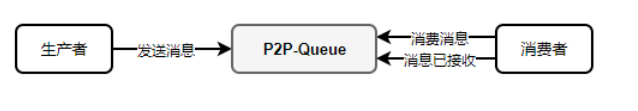
**5 点对点(p2p)模式**

**点对点(p2p)模式有三个角色：消息队列（Queue），发送者(Sender)，接收者(Receiver)。**发送者将消息发送到一个特定的队列中，等待接收者从队列中获取消息消耗。

一、每个消息只能被一个接收者消费，且消息被消费后默认从队列中删掉（也可以通过其他签收机制重复消费）。

二、发送者和接收者之间没有依赖性，生产者发送消息和消费者接收消息并不要求同时运行。

三、接收者在成功接收消息之后需向队列发送接收成功的确认消息。



**6 发布订阅(Pub/Sub)模式**

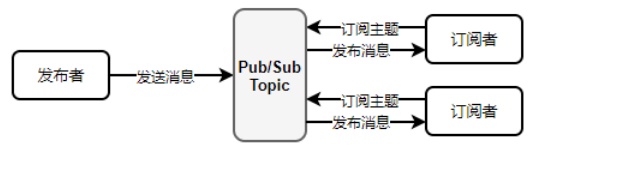
发布订阅(Pub/Sub)模式也有三个角色：主题（Topic），发布者（Publisher），订阅者（Subscriber）。发布者将消息发送到主题队列中，系统再将这些消息传递给订阅者。

一、每个消息可以被多个订阅者消费。

二、发布者和订阅者之间存在依赖性。订阅者必须先订阅主题后才能接收到信息，在订阅前发布的消息，订阅者是接收不到的。

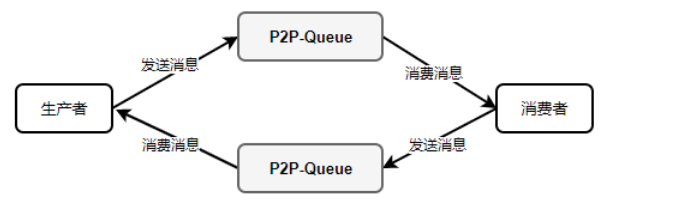
三、**非持久化订阅**：如果订阅者不在线，此时发布的消息订阅者是也接收不到，即便订阅者重新上线也接收不到。

四、**持久化订阅：**订阅者订阅主题后，即便订阅者不在线，此时发布的消息可以在订阅者重新上线后接收到的。



**7 双向应答模式**

双向应答模式并不是消息中间件提供的一种通信模式，它是由于实际生成环境的需要，在原有的基础上做了改良。即**消息的发送者也是消息的接收者。消息的接收者也是消息的发送者。**



**8 为什么需要消息系统**

**（1）.解耦：**

　　允许你独立的扩展或修改两边的处理过程，只要确保它们遵守同样的接口约束。

**（2）.冗余：**

　　消息队列把数据进行持久化直到它们已经被完全处理，通过这一方式规避了数据丢失风险。许多消息队列所采用的"插入-获取-删除"范式中，在把一个消息从队列中删除之前，需要你的处理系统明确的指出该消息已经被处理完毕，从而确保你的数据被安全的保存直到你使用完毕。

**（3）.扩展性**：

　　因为消息队列解耦了你的处理过程，所以增大消息入队和处理的频率是很容易的，只要另外增加处理过程即可。

**（4）.灵活性 & 峰值处理能力**：

　　在访问量剧增的情况下，应用仍然需要继续发挥作用，但是这样的突发流量并不常见。如果为以能处理这类峰值访问为标准来投入资源随时待命无疑是巨大的浪费。使用消息队列能够使关键组件顶住突发的访问压力，而不会因为突发的超负荷的请求而完全崩溃。

**（5）.可恢复性**：

　　系统的一部分组件失效时，不会影响到整个系统。消息队列降低了进程间的耦合度，所以即使一个处理消息的进程挂掉，加入队列中的消息仍然可以在系统恢复后被处理。

**（6）.顺序保证**：

　　在大多使用场景下，数据处理的顺序都很重要。大部分消息队列本来就是排序的，并且能保证数据会按照特定的顺序来处理。（Kafka 保证一个 Partition 内的消息的有序性）

**（7）.缓冲：**

　　有助于控制和优化数据流经过系统的速度，解决生产消息和消费消息的处理速度不一致的情况。

**（8）.异步通信：**

　　很多时候，用户不想也不需要立即处理消息。消息队列提供了异步处理机制，允许用户把一个消息放入队列，但并不立即处理它。想向队列中放入多少消息就放多少，然后在需要的时候再去处理它们。

**9 常用Message Queue对比**

**RabbitMQ**

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**多协议、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。

**Redis**

Redis是一个基于Key-Value对的NoSQL数据库，开发维护很活跃。虽然它是一个Key-Value数据库存储系统，但它本身支持MQ功能，所以完全可以当做一个轻量级的队列服务来使用。对于RabbitMQ和Redis的入队和出队操作，各执行100万次，每10万次记录一次执行时间。测试数据分为128Bytes、512Bytes、1K和10K四个不同大小的数据。实验表明：入队时，当数据比较小时Redis的性能要高于RabbitMQ，而如果数据大小超过了10K，Redis则慢的无法忍受；出队时，无论数据大小，Redis都表现出非常好的性能，而RabbitMQ的出队性能则远低于Redis。

**ZeroMQ**

ZeroMQ号称最快的消息队列系统，尤其针对大吞吐量的需求场景。ZMQ能够实现RabbitMQ不擅长的**高级/复杂的队列**，但是开发人员需要自己组合多种技术框架，技术上的复杂度是对这MQ能够应用成功的挑战。ZeroMQ具有一个独特的非中间件的模式，你不需要安装和运行一个消息服务器或中间件，因为你的应用程序将扮演了这个服务角色。你只需要简单的引用ZeroMQ程序库，可以使用NuGet安装，然后你就可以愉快的在应用程序之间发送消息了。但是ZeroMQ仅提供非持久性的队列，也就是说如果宕机，数据将会丢失。其中，Twitter的Storm 0.9.0以前的版本中默认使用ZeroMQ作为数据流的传输（Storm从0.9版本开始同时支持ZeroMQ和Netty作为传输模块）。

**ActiveMQ**

ActiveMQ是**Apache下java的一个子项目**。 **支持事务**。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。

**Kafka**

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，而Jafka是在Kafka之上孵化而来的，即Kafka的一个升级版。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。Kafka通过Hadoop的并行加载机制来统一了在线和离线的消息处理。Apache Kafka相对于ActiveMQ是一个非常轻量级的消息系统，除了性能非常好之外，还是一个工作良好的分布式系统。



## 二、ActiveMQ

**1 JMS和MQ**

JMS是一种与厂商无关的 API，用来访问消息收发系统消息。它类似于JDBC(Java DatabaseConnectivity)：这里，JDBC 是可以用来访问许多不同关系数据库的 API，而 JMS 则提供同样与厂商无关的访问方法，以访问消息收发服务。许多厂商目前都支持JMS，包括 IBM 的 MQSeries、BEA的 Weblogic JMS service和 Progress 的 SonicMQ，这只是几个例子。 JMS 使您能够通过消息收发服务（有时称为消息中介程序或路由器）从一个 JMS 客户机向另一个JMS客户机发送消息。消息是 JMS 中的一种类型对象，由两部分组成：报头和消息主体。报头由路由信息以及有关该消息的元数据组成。消息主体则携带着应用程序的数据或有效负载。根据有效负载的类型来划分，可以将消息分为几种类型，它们分别携带：简单文本(TextMessage)、可序列化的对象 (ObjectMessage)、属性集合 (MapMessage)、字节流 (BytesMessage)、原始值流 (StreamMessage)，还有无有效负载的消息 (Message)。目前选择的最多的是ActiveMQ。

**JMS是一个用于提供消息服务的技术规范，它制定了在整个消息服务提供过程中的所有数据结构和交互流程。而MQ则是消息队列服务，是面向消息中间件（MOM）的最终实现，是真正的服务提供者；MQ的实现可以基于JMS，也可以基于其他规范或标准。**

**ActiveMQ 是Apache出品，最流行的，能力强劲的开源消息总线。ActiveMQ 是一个完全支持JMS1.1和J2EE 1.4规范的 JMS Provider实现,尽管JMS规范出台已经是很久的事情了,但是JMS在当今的J2EE应用中间仍然扮演着特殊的地位。**

**2 ActiveMQ主要特点**

1. 多种语言和协议编写客户端。语言: Java, C, C++, C#, Ruby, Perl, Python, PHP。应用协议: OpenWire,Stomp REST,WSNotification,XMPP,AMQP

2. 完全支持JMS1.1和J2EE 1.4规范 (持久化,XA消息,事务)

3. 对Spring的支持,ActiveMQ可以很容易内嵌到使用Spring的系统里面去,而且也支持Spring2.0的特性

4. 通过了常见J2EE服务器(如 Geronimo,JBoss 4, GlassFish,WebLogic)的测试,其中通过JCA 1.5 resource adaptors的配置,可以让ActiveMQ可以自动的部署到任何兼容J2EE 1.4 商业服务器上

5. 支持多种传送协议:in-VM,TCP,SSL,NIO,UDP,JGroups,JXTA

6. 支持通过JDBC和journal提供高速的消息持久化

7. 从设计上保证了高性能的集群,客户端-服务器,点对点

8. 支持Ajax

9. 支持与Axis的整合

10. 可以很容易得调用内嵌JMS provider,进行测试

11. ActiveMQ速度非常快；一般要比jbossMQ快10倍。

**3 ActiveMQ优点**

一个快速的开源消息组件(框架)，支持集群，同等网络，自动检测，TCP，SSL，广播，持久化，XA，和J2EE1.4容器无缝结合，并且支持轻量级容器和大多数跨语言客户端上的Java虚拟机。消息异步接受，减少软件多系统集成的耦合度。消息可靠接收，确保消息在中间件可靠保存，多个消息也可以组成原子事务。

**4 ActiveMQ缺点**

一、ActiveMQ默认的配置性能偏低，需要优化配置，但是配置文件复杂，ActiveMQ本身不提供管理工具；示例代码少；主页上的文档看上去比较全面，但是缺乏一种有效的组织方式，文档只有片段，用户很难由浅入深进行了解，

二、文档整体的专业性太强。在研究阶段可以通过查maillist、看Javadoc、分析源代码来了解。

**5 ActiveMQ应用场景**

**1、 不同语言应用集成**

ActiveMQ 中间件用Java语言编写，因此自然提供Java客户端 API。但是ActiveMQ 也为C/C++、.NET、Perl、PHP、Python、Ruby 和一些其它语言提供客户端。在你考虑如何集成不同平台不同语言编写应用的时候，ActiveMQ 拥有巨大优势。在这样的例子中，多种客户端API通过ActiveMQ 发送和接受消息成为可能，无论使用的是什么语言。此外，ActiveMQ 还提供交叉语言功能，该功能整合这种功能，无需使用远程过程调用（RPC）确实是个优势，因为消息协助应用解耦。

**2、 作为RPC的替代**

使用RPC同步调用的应用十分普遍。假设大多数客户端服务器应用使用RPC，包括ATM、大多数WEB应用、信用卡系统、销售点系统等等。尽管很多系统很成功，但是转换使用异步消息可以带来很多好处，而且也不会放弃响应保证。使用同步请求的系统在规模上有较大的限制，因为请求会被阻塞，从而导致整个系统变慢。如果使用异步消息替代，可以很容易增加额外的消息接收者，使得消息能被并发消耗，从而加快请求处理。当然，你的系统应用间应该是解耦的。

**3、 应用之间解耦**

正如之前讨论的，紧耦合架构可以导致很多问题，尤其是如果他们是分布的。松耦合架构，在另一方面，证实了更少的依赖性，能够更好地处理不可预见的改变。不仅可以在系统中改变组件而不影响整个系统，而且组件交互也相当的简单。相比使用同步的系统（调用者必须等待被调用者返回信息），异步系统（调用方发送消息后就不管，即fire-and-forget）能够给我们带来事件驱动架构（event-driven architecture EDA）。

**4、 作为事件驱动架构的主干**

解耦，异步架构的系统允许通过代理器自己配置更多的客户端，内存等（即vertical scalability）来扩大系统，而不是增加更多的代理器（即horizontal scalability）。考虑如亚马逊这样繁忙的电子商务系统。当用户购买物品，事实上系统需要很多步骤去处理，包括下单，创建发票，付款，执行订单，运输等。但是用户下单后，会立即返回“谢谢你下单”的界面。不只是没有延迟，而且用户还会受到一封邮件表明订单已经收到。在亚马逊下单的例子就是一个多步处理的例子。每一步都由单独的服务去处理。当用户下单是，有一个同步的体积表单动作，但整个处理流程并不通过浏览器同步处理。相反地，订单马上被接受和反馈。而剩下的步骤就通过异步处理。如果在处理过程中出错，用户会通过邮件收到通知。这样的异步处理能提供高负载和高可用性。

**5、 提高系统扩展性**

很多使用事件驱动设计的系统是为了获得高可扩展性，例如电子商务，政府，制造业，线上游戏等。通过异步消息分开商业处理步骤给各个应用，能够带来很多可能性。考虑设计一个应用来完成一项特殊的任务。这就是面向服务的架构（service-oriented architecture SOA）。每一个服务完成一个功能并且只有一个功能。应用就通过服务组合起来，服务间使用异步消息和最终一致性。这样的设计便可以引入一个复杂事件处理概念（complex event processing CEP）。使用CEP，部件间的交互可以被记录追踪。在异步消息系统中，可以很容易在部件间增加一层处理。

**6 下载安装ActiveMQ**

ActiveMQ官网下载地址：http://activemq.apache.org/download.html

ActiveMQ 提供了Windows 和Linux、Unix 等几个版本，楼主这里选择了Linux 版本下进行开发。下载完安装包，解压之后的目录。

**从它的目录来说，还是很简单的：**

bin存放的是脚本文件

conf存放的是基本配置文件

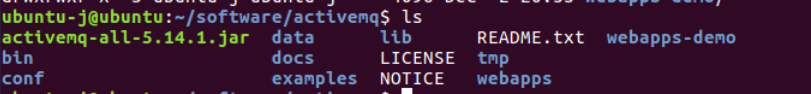
data存放的是日志文件

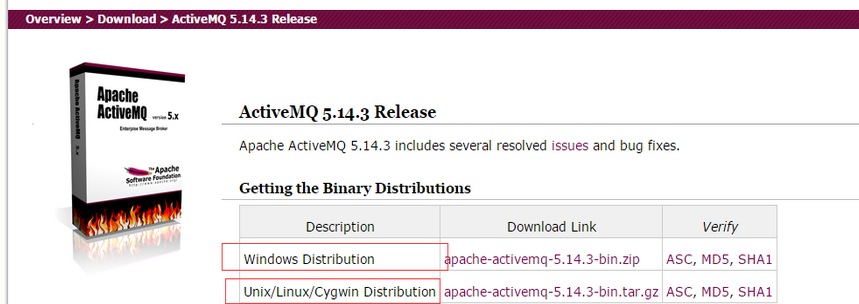
docs存放的是说明文档

examples存放的是简单的实例

lib存放的是activemq所需jar包

webapps用于存放项目的目录

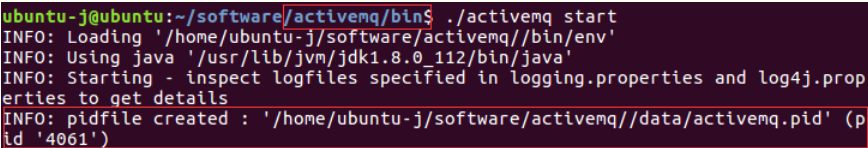




**7 启动ActiveMQ**

进入到ActiveMQ 安装目录的Bin 目录，linux 下输入 ./activemq start 启动activeMQ 服务。

输入命令之后，会提示我们创建了一个进程IP 号，这时候说明服务已经成功启动了。



ActiveMQ默认启动时，启动了内置的jetty服务器，提供一个用于监控ActiveMQ的admin应用。

**admin：http://127.0.0.1:8161/admin/**

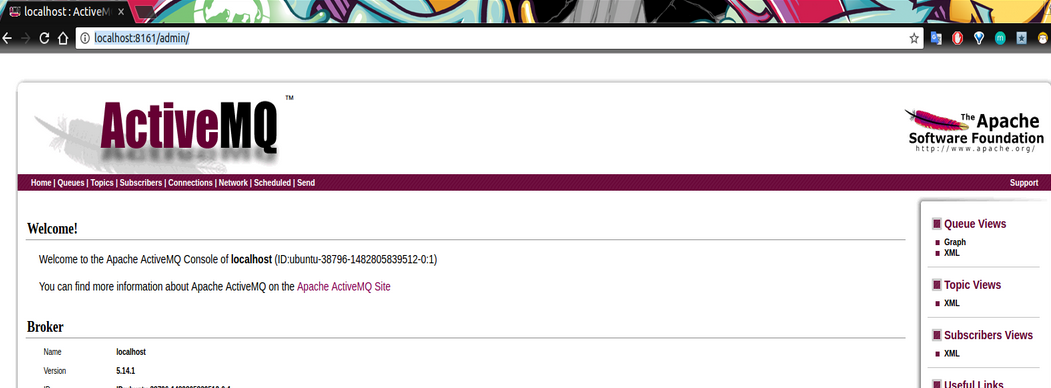
我们在浏览器打开链接之后输入账号密码（这里和tomcat 服务器类似）

默认账号：admin

密码：admin

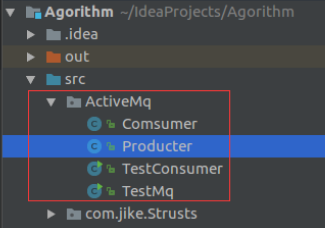
到这里为止，ActiveMQ 服务端就启动完毕了。

ActiveMQ 在linux 下的终止命令是 ./activemq stop



**8 创建一个ActiveMQ工程**

项目目录结构：



上述在官网下载ActiveMq 的时候，我们可以在目录下看到一个jar包：

这个jar 包就是我们需要在项目中进行开发中使用到的相关依赖。

**3.1 创建生产者**

public class Producter {

//ActiveMq 的默认用户名

private static final String USERNAME = ActiveMQConnection.DEFAULT\_USER;

//ActiveMq 的默认登录密码

private static final String PASSWORD = ActiveMQConnection.DEFAULT\_PASSWORD;

//ActiveMQ 的链接地址

private static final String BROKEN\_URL = ActiveMQConnection.DEFAULT\_BROKER\_URL;

AtomicInteger count = new AtomicInteger(0);

//链接工厂

ConnectionFactory connectionFactory;

//链接对象

Connection connection;

//事务管理

Session session;

ThreadLocal<MessageProducer> threadLocal = new ThreadLocal<>();

public void init(){

try {

//创建一个链接工厂

connectionFactory = new ActiveMQConnectionFactory(USERNAME,PASSWORD,BROKEN\_URL);

//从工厂中创建一个链接

connection = connectionFactory.createConnection();

//开启链接

connection.start();

//创建一个事务（这里通过参数可以设置事务的级别）

session = connection.createSession(true,Session.SESSION\_TRANSACTED);

} catch (JMSException e) {

e.printStackTrace();

}

}

public void sendMessage(String disname){

try {

//创建一个消息队列

Queue queue = session.createQueue(disname);

//消息生产者

MessageProducer messageProducer = null;

if(threadLocal.get()!=null){

messageProducer = threadLocal.get();

}else{

messageProducer = session.createProducer(queue);

threadLocal.set(messageProducer);

}

while(true){

Thread.sleep(1000);

int num = count.getAndIncrement();

//创建一条消息

TextMessage msg = session.createTextMessage(Thread.currentThread().getName()+

"productor:我是大帅哥，我现在正在生产东西！,count:"+num);

System.out.println(Thread.currentThread().getName()+

"productor:我是大帅哥，我现在正在生产东西！,count:"+num);

//发送消息

messageProducer.send(msg);

//提交事务

session.commit();

}

} catch (JMSException e) {

e.printStackTrace();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

**3.2 创建消费者**

public class Comsumer {

private static final String USERNAME = ActiveMQConnection.DEFAULT\_USER;

private static final String PASSWORD = ActiveMQConnection.DEFAULT\_PASSWORD;

private static final String BROKEN\_URL = ActiveMQConnection.DEFAULT\_BROKER\_URL;

ConnectionFactory connectionFactory;

Connection connection;

Session session;

ThreadLocal<MessageConsumer> threadLocal = new ThreadLocal<>();

AtomicInteger count = new AtomicInteger();

public void init(){

try {

connectionFactory = new ActiveMQConnectionFactory(USERNAME,PASSWORD,BROKEN\_URL);

connection = connectionFactory.createConnection();

connection.start();

session = connection.createSession(false,Session.AUTO\_ACKNOWLEDGE);

} catch (JMSException e) {

e.printStackTrace();

}

}

public void getMessage(String disname){

try {

Queue queue = session.createQueue(disname);

MessageConsumer consumer = null;

if(threadLocal.get()!=null){

consumer = threadLocal.get();

}else{

consumer = session.createConsumer(queue);

threadLocal.set(consumer);

}

while(true){

Thread.sleep(1000);

TextMessage msg = (TextMessage) consumer.receive();

if(msg!=null) {

msg.acknowledge();

System.out.println(Thread.currentThread().getName()+": Consumer:我是消费者，我正在消费Msg"+msg.getText()+"--->"+count.getAndIncrement());

}else {

break;

}

}

} catch (JMSException e) {

e.printStackTrace();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

**9 运行ActiveMQ项目**

**4.1 生产者开始生产消息**

public class TestMq {

public static void main(String[] args){

Producter producter = new Producter();

producter.init();

TestMq testMq = new TestMq();

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

//Thread 1

new Thread(testMq.new ProductorMq(producter)).start();

//Thread 2

new Thread(testMq.new ProductorMq(producter)).start();

//Thread 3

new Thread(testMq.new ProductorMq(producter)).start();

//Thread 4

new Thread(testMq.new ProductorMq(producter)).start();

//Thread 5

new Thread(testMq.new ProductorMq(producter)).start();

}

private class ProductorMq implements Runnable{

Producter producter;

public ProductorMq(Producter producter){

this.producter = producter;

}

@Override

public void run() {

while(true){

try {

producter.sendMessage("Jaycekon-MQ");

Thread.sleep(10000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

}

**运行结果：**

INFO | Successfully connected to tcp://localhost:61616

Thread-6productor:我是大帅哥，我现在正在生产东西！,count:0

Thread-4productor:我是大帅哥，我现在正在生产东西！,count:1

Thread-2productor:我是大帅哥，我现在正在生产东西！,count:3

Thread-5productor:我是大帅哥，我现在正在生产东西！,count:2

Thread-3productor:我是大帅哥，我现在正在生产东西！,count:4

Thread-6productor:我是大帅哥，我现在正在生产东西！,count:5

Thread-3productor:我是大帅哥，我现在正在生产东西！,count:6

Thread-5productor:我是大帅哥，我现在正在生产东西！,count:7

Thread-2productor:我是大帅哥，我现在正在生产东西！,count:8

Thread-4productor:我是大帅哥，我现在正在生产东西！,count:9

Thread-6productor:我是大帅哥，我现在正在生产东西！,count:10

Thread-3productor:我是大帅哥，我现在正在生产东西！,count:11

Thread-5productor:我是大帅哥，我现在正在生产东西！,count:12

Thread-2productor:我是大帅哥，我现在正在生产东西！,count:13

Thread-4productor:我是大帅哥，我现在正在生产东西！,count:14

Thread-6productor:我是大帅哥，我现在正在生产东西！,count:15

Thread-3productor:我是大帅哥，我现在正在生产东西！,count:16

Thread-5productor:我是大帅哥，我现在正在生产东西！,count:17

Thread-2productor:我是大帅哥，我现在正在生产东西！,count:18

Thread-4productor:我是大帅哥，我现在正在生产东西！,count:19

**4.2 消费者开始消费消息**

public class TestConsumer {

public static void main(String[] args){

Comsumer comsumer = new Comsumer();

comsumer.init();

TestConsumer testConsumer = new TestConsumer();

new Thread(testConsumer.new ConsumerMq(comsumer)).start();

new Thread(testConsumer.new ConsumerMq(comsumer)).start();

new Thread(testConsumer.new ConsumerMq(comsumer)).start();

new Thread(testConsumer.new ConsumerMq(comsumer)).start();

}

private class ConsumerMq implements Runnable{

Comsumer comsumer;

public ConsumerMq(Comsumer comsumer){

this.comsumer = comsumer;

}

@Override

public void run() {

while(true){

try {

comsumer.getMessage("Jaycekon-MQ");

Thread.sleep(10000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

}

**运行结果：**

INFO | Successfully connected to tcp://localhost:61616

Thread-2: Consumer:我是消费者，我正在消费MsgThread-5productor:我是大帅哥，我现在正在生产东西！,count:4--->0

Thread-3: Consumer:我是消费者，我正在消费MsgThread-4productor:我是大帅哥，我现在正在生产东西！,count:36--->1

Thread-4: Consumer:我是消费者，我正在消费MsgThread-3productor:我是大帅哥，我现在正在生产东西！,count:38--->2

Thread-5: Consumer:我是消费者，我正在消费MsgThread-6productor:我是大帅哥，我现在正在生产东西！,count:37--->3

Thread-2: Consumer:我是消费者，我正在消费MsgThread-6productor:我是大帅哥，我现在正在生产东西！,count:2--->4

Thread-3: Consumer:我是消费者，我正在消费MsgThread-5productor:我是大帅哥，我现在正在生产东西！,count:40--->5

Thread-4: Consumer:我是消费者，我正在消费MsgThread-6productor:我是大帅哥，我现在正在生产东西！,count:42--->6

Thread-5: Consumer:我是消费者，我正在消费MsgThread-4productor:我是大帅哥，我现在正在生产东西！,count:41--->7

Thread-2: Consumer:我是消费者，我正在消费MsgThread-3productor:我是大帅哥，我现在正在生产东西！,count:1--->8

Thread-3: Consumer:我是消费者，我正在消费MsgThread-2productor:我是大帅哥，我现在正在生产东西！,count:44--->9

Thread-4: Consumer:我是消费者，我正在消费MsgThread-4productor:我是大帅哥，我现在正在生产东西！,count:46--->10

Thread-5: Consumer:我是消费者，我正在消费MsgThread-5productor:我是大帅哥，我现在正在生产东西！,count:45--->11

Thread-2: Consumer:我是消费者，我正在消费MsgThread-2productor:我是大帅哥，我现在正在生产东西！,count:3--->12

Thread-3: Consumer:我是消费者，我正在消费MsgThread-3productor:我是大帅哥，我现在正在生产东西！,count:48--->13

Thread-4: Consumer:我是消费者，我正在消费MsgThread-5productor:我是大帅哥，我现在正在生产东西！,count:50--->14

Thread-5: Consumer:我是消费者，我正在消费MsgThread-2productor:我是大帅哥，我现在正在生产东西！,count:49--->15

Thread-4: Consumer:我是消费者，我正在消费MsgThread-2productor:我是大帅哥，我现在正在生产东西！,count:54--->16

Thread-2: Consumer:我是消费者，我正在消费MsgThread-5productor:我是大帅哥，我现在正在生产东西！,count:6--->17

Thread-3: Consumer:我是消费者，我正在消费MsgThread-6productor:我是大帅哥，我现在正在生产东西！,count:52--->18

Thread-5: Consumer:我是消费者，我正在消费MsgThread-3productor:我是大帅哥，我现在正在生产东西！,count:53--->19

Thread-4: Consumer:我是消费者，我正在消费MsgThread-3productor:我是大帅哥，我现在正在生产东西！,count:58--->20

## 三、RabbitMQ

**1 RabbitMQ 是一个由 Erlang 语言开发的 AMQP 的开源实现。**

AMQP ：Advanced Message Queue，高级消息队列协议。它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，并不受产品、开发语言等条件的限制。

**RabbitMQ 最初起源于金融系统，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。**

**2 RabbitMQ 特点**

@可靠性（Reliability）RabbitMQ 使用一些机制来保证可靠性，如持久化、传输确认、发布确认。

@灵活的路由（Flexible Routing）在消息进入队列之前，通过 Exchange 来路由消息的。对于典型的路由功能，RabbitMQ 已经提供了一些内置的 Exchange 来实现。针对更复杂的路由功能，可以将多个 Exchange 绑定在一起，也通过插件机制实现自己的 Exchange 。

@消息集群（Clustering）多个 RabbitMQ 服务器可以组成一个集群，形成一个逻辑 Broker 。

@高可用（Highly Available Queues）队列可以在集群中的机器上进行镜像，使得在部分节点出问题的情况下队列仍然可用。

@多种协议（Multi-protocol）RabbitMQ 支持多种消息队列协议，比如 STOMP、MQTT 等等。

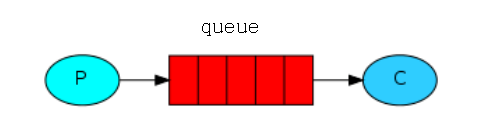
@多语言客户端（Many Clients）RabbitMQ 几乎支持所有常用语言，比如 Java、.NET、Ruby 等等。

@管理界面（Management UI）RabbitMQ 提供了一个易用的用户界面，使得用户可以监控和管理消息 Broker 的许多方面。

@跟踪机制（Tracing）如果消息异常，RabbitMQ 提供了消息跟踪机制，使用者可以找出发生了什么。

@插件机制（Plugin System）RabbitMQ 提供了许多插件，来从多方面进行扩展，也可以编写自己的插件。

**3 RabbitMQ 中的概念模型**

MQ 产品从模型抽象上来说都是一样的过程：消费者（consumer）订阅某个队列。生产者（producer）创建消息，然后发布到队列（queue）中，最后将消息发送到监听的消费者。

**4 RabbitMQ 内部结构**

**1 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。

**2 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。

**3 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。

**4 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。

**5 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。

**6 Connection 网络连接**，比如一个TCP连接。

**7 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连接内地虚拟连接，AMQP命令都是通过信道发出去的，

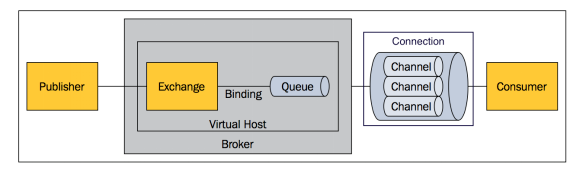
不管是发布消息、订阅队列还是接收消息，这些动作都是通过信道完成。因为对于操作系统

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。

**8Consumer 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。

**9 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 / 。

**10 Broker 表示消息队列服务器实体。**



**5 RabbitMQ一些知识点**

**--AMQP协议**

AMQP协议是一个提供统一消息服务的应用层标准协议，并不会受到客户端／中间件不同产品、不同开发语言等条件的影响。RabbitMQ则是基于该协议实现的。

举个例子来说，如下图，生产者将消息发送到交换机上，交换机接收到信息以后按照相应的路由键路由到队列中，这里的交换机只是起到了路由的功能，实际

上消息存储在消息队列中，接下来在队列中的的消息可以将其散发到消费者或者消费者主动去获取消息。

**--三层协议**

该协议分为三层，分别是Module Layer、Session Layer、Transport Layer，如下图：

@Module Layer：决定基本域模型所产生的行为，主要定义了一些供客户端调用的命令，客户端可以利用这些命令实现自己的业务逻辑。

@Session Layer：主要负责将客户端的命令发送给服务器，再将服务端的应答返回给客户端，主要为客户端和服务器之间通信提供可靠性、同步机制和错误处理。

@Transport Layer：主要用于二进制数据流的传输。

**--RabbitMQ名词**

Broker：接收和分发消息的应用，像是RabbitMQ、ZeroMQ或者Redis等等。

Connection：publisher/consumer和Broker之间的TCP连接

Channel：两个AMQP结点之间双向通信流，通道是多路复用的，因此单个网络连接可以支撑多个通道，各个通道之间是相互隔离的，其极大的减少了操作系统建立多个TCP连接的开销。

Message：消息

Exchange：服务器中接收来自生产者程序的消息的实体，并可选择将这些消息路由到服务器中的消息队列中

Message queue：保存消息并将它们转发给消费者

Routing key：一个虚拟地址，虚拟机可用它来确定如何路由一个特定消息

**--交换机属性**

Name (名称)

Durability (消息代理重启后，交换机是否还存在)

Auto-delete (当所有与此交换机绑定的队列都不再使用此交换机时是否自动删除)

**--交换机类型**

Exchange存在多种类型，其中最常使用的包括Direct、Fanout、Topic三种类型，还有一种特殊的Headers，它们分别的用处是：

（1）Direct：Message中的routing key 如果和Binding中的 binding key一致的话，则Exchange会将Message散发到对应的Message queue中。

在默认的情况下创建的就是该类型的exchange，通常使用在需要发送消息到具体的队列的情况，比如下面这张图中，最后交换机会将消息路由到名字为green的消息队列中。

（2）Fanout：Exchange会将所有的Message散发到关联的Message queue中。

即便是已经提供了routingkey，该类型交换机也会忽略掉，并且将消息发送到所有关联的队列中，通常用来实现发布／订阅模式，比如说体育新闻网站可以用它来近乎实时地

将比分更新分发给移动客户端，分发系统使用它来广播各种状态和配置更新等等。

（3）Topic：根据正则表达式将Message散发到匹配的Message queue中。

比如在下图中，根据Routing key first.green.fast 能够匹配到的队列有.green.和.\*.fast，该模式主要用来实现消息多播路由。比如说分发有关于特定地理位置的数据，例如销售点，涉及到分类或者标签的新闻更新（例如，针对特定的运动项目或者队伍）等等。

（4）Headers：类似于Direct，但是当涉及到多个属性的时候，一个Routing key 无法完全表达，其只能是个字符串，而多属性需要使用到多个键值对表示，因此使用消息属性来代替路由键作为路由规则，通过判断消息头中的值来和队列中的值来确定消息的路由位置。

如果x-match为any的时候， 表示消息头的任意一个值被匹配后就可以满足条件，而当x-match为 all的时候，表示消息头的所有的值都需要被匹配到才能满足条件，比如在下图中，因为第一个队列为any，并且存在匹配的值，所以可以路由到绿色的路由中，因为第三个队列为all，消息头并没有都匹配所有的键值对，因此消息只会发送到绿色的消息队列中。

**--消息存在三个动作，分别是确认、拒绝、预取**

（1）消息确认

消息确认存在两种模式，分别是1. 自动确认模式：消息队列发送完后就销毁；2.显示确认模式：在消费者进行确认后再进行销毁；

在显示确认模式情况下，如果消费者不幸挂掉了，该机制会使消息重新回到消息队列中，等待下一位消费者。

（2）拒绝消息

分为重新放回队列还是立即销毁，像是在kombo中，它们分别通过message.requeue()和message.reject()实现。

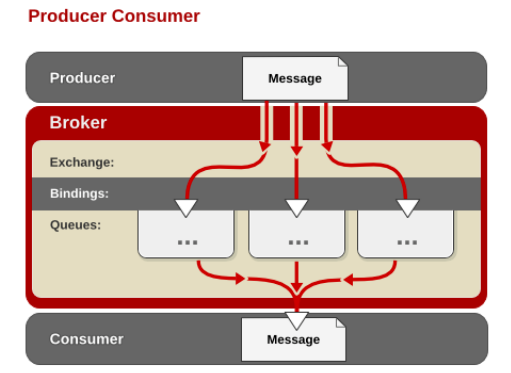
（3）预取

如果在多个消费者共享一个消息队列的情况下，可以在消费者发送消息的时候设置预取的消息个数，起到一个负载均衡的效果。

连接和通道连接即Connection，内部则是通过TCP长连接实现，关闭连接时最好优雅的关闭AMQP连接，即connection.close()，而不是简单粗暴的关闭TCP连接，否则会报出相关的连接错误信息。

**6 AMQP 中的消息路由**

AMQP 中消息的路由过程和 Java 开发者熟悉的 JMS 存在一些差别，AMQP 中增加了 Exchange 和 Binding 的角色。生产者把消息发布到 Exchange 上，消息最终到达队列并被消费者接收，而 Binding 决定交换器的消息应该发送到那个队列。



**7 Exchange 类型**

Exchange分发消息时根据类型的不同分发策略有区别，目前共四种类型：direct、fanout、topic、headers 。headers 匹配 AMQP 消息的 header 而不是路由键，此外 headers 交换器和 direct 交换器完全一致，但性能差很多，目前几乎用不到了，所以直接看另外三种类型：

**1 direct 交换器**

消息中的路由键（routing key）如果和 Binding 中的 binding key 一致， 交换器就将消息发到对应的队列中。路由键与队列名完全匹配，如果一个队列绑定到交换机要求路由键为“dog”，则只转发 routing key 标记为“dog”的消息，不会转发“dog.puppy”，也不会转发“dog.guard”等等。它是完全匹配、单播的模式。

**2 fanout 交换器**

每个发到 fanout 类型交换器的消息都会分到所有绑定的队列上去。fanout 交换器不处理路由键，只是简单的将队列绑定到交换器上，每个发送到交换器的消息都会被转发到与该交换器绑定的所有队列上。很像子网广播，每台子网内的主机都获得了一份复制的消息。fanout 类型转发消息是最快的。

**3 topic 交换器**

交换器通过模式匹配分配消息的路由键属性，将路由键和某个模式进行匹配，此时队列需要绑定到一个模式上。它将路由键和绑定键的字符串切分成单词，这些单词之间用点隔开。它同样也会识别两个通配符：符号“#”和符号“”。#匹配0个或多个单词，匹配不多不少一个单词。

**8 RabbitMQ 安装**

一般来说安装 RabbitMQ 之前要安装 Erlang ，可以去Erlang官网下载。接着去RabbitMQ官网下载安装包，之后解压缩即可。根据操作系统不同官网提供了相应的安装说明：Windows、Debian / Ubuntu、RPM-based Linux、Mac。

更新 brew：brew update

安装 rabbitmq 服务器：brew install rabbitmq

这样 RabbitMQ 就安装好了，安装过程中会自动其所依赖的 Erlang 。

**9 RabbitMQ 运行和管理**

启动很简单

找到安装后的 RabbitMQ 所在目录下的 sbin 目录

./sbin/rabbitmq-server

启动正常的话会看到一些启动过程信息和最后的 completed with 7 plugins，这也说明启动的时候默认加载了7个插件。

正常启动

后台启动如果想让 RabbitMQ 以守护程序的方式在后台运行，可以在启动的时候加上 -detached 参数：

./sbin/rabbitmq-server -detached

查询服务器状态

sbin 目录下有个特别重要的文件叫 rabbitmqctl ，它提供了 RabbitMQ 管理需要的几乎一站式解决方案，绝大部分的运维命令它都可以提供。

查询 RabbitMQ 服务器的状态信息可以用参数 status ：./sbin/rabbitmqctl status

关闭 RabbitMQ 节点：./sbin/rabbitmqctl stop

它会和本地节点通信并指示其干净的关闭，也可以指定关闭不同的节点，包括远程节点，只需要传入参数 -n ：

./sbin/rabbitmqctl -n rabbit@server.example.com stop

关闭 RabbitMQ 应用程序

如果只想关闭应用程序，同时保持 Erlang 节点运行则可以用 stop\_app：

./sbin/rabbitmqctl stop\_app

启动 RabbitMQ 应用程序：./sbin/rabbitmqctl start\_app

重置 RabbitMQ 节点：./sbin/rabbitmqctl reset

**10 Java 客户端访问**

**maven工程的pom文件中添加依赖**

<dependency>

<groupId>com.rabbitmq</groupId>

<artifactId>amqp-client</artifactId>

<version>4.1.0</version>

</dependency>

**消息生产者**

package org.study.rabbitmq;

import com.rabbitmq.client.Channel;

import com.rabbitmq.client.Connection;

import com.rabbitmq.client.ConnectionFactory;

import java.io.IOException;

import java.util.concurrent.TimeoutException;

public class Producer {

public static void main(String[] args) throws IOException, TimeoutException {

//创建连接工厂

ConnectionFactory factory = new ConnectionFactory();

factory.setUsername("guest");

factory.setPassword("guest");

//设置 RabbitMQ 地址

factory.setHost("localhost");

//建立到代理服务器到连接

Connection conn = factory.newConnection();

//获得信道

Channel channel = conn.createChannel();

//声明交换器

String exchangeName = "hello-exchange";

channel.exchangeDeclare(exchangeName, "direct", true);

String routingKey = "hola";

//发布消息

byte[] messageBodyBytes = "quit".getBytes();

channel.basicPublish(exchangeName, routingKey, null, messageBodyBytes);

channel.close();

conn.close();

}

}

**消息消费者**

package org.study.rabbitmq;

import com.rabbitmq.client.\*;

import java.io.IOException;

import java.util.concurrent.TimeoutException;

public class Consumer {

public static void main(String[] args) throws IOException, TimeoutException {

ConnectionFactory factory = new ConnectionFactory();

factory.setUsername("guest");

factory.setPassword("guest");

factory.setHost("localhost");

//建立到代理服务器到连接

Connection conn = factory.newConnection();

//获得信道

final Channel channel = conn.createChannel();

//声明交换器

String exchangeName = "hello-exchange";

channel.exchangeDeclare(exchangeName, "direct", true);

//声明队列

String queueName = channel.queueDeclare().getQueue();

String routingKey = "hola";

//绑定队列，通过键 hola 将队列和交换器绑定起来

channel.queueBind(queueName, exchangeName, routingKey);

while(true) {

//消费消息

boolean autoAck = false;

String consumerTag = "";

channel.basicConsume(queueName, autoAck, consumerTag, new DefaultConsumer(channel) {

@Override

public void handleDelivery(String consumerTag,

Envelope envelope,

AMQP.BasicProperties properties,

byte[] body) throws IOException {

String routingKey = envelope.getRoutingKey();

String contentType = properties.getContentType();

System.out.println("消费的路由键：" + routingKey);

System.out.println("消费的内容类型：" + contentType);

long deliveryTag = envelope.getDeliveryTag();

//确认消息

channel.basicAck(deliveryTag, false);

System.out.println("消费的消息体内容：");

String bodyStr = new String(body, "UTF-8");

System.out.println(bodyStr);

}

});

}

}

}

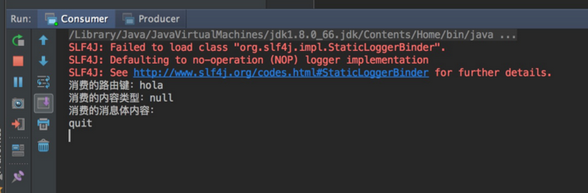
**启动 RabbitMQ 服务器**

./sbin/rabbitmq-server

先运行 Consumer ，这样当生产者发送消息的时候能在消费者后端看到消息记录。

接着运行 Producer ,发布一条消息，在 Consumer 的控制台能看到接收的消息：

**Consumer 控制台**



**11 RabbitMQ 集群**

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。

**1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

**2交换器元数据**

交换器名称、类型、属性

**3绑定元数据**

内部是一张表格记录如何将消息路由到队列

**4vhost 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

**12 RabbitMQ 集群配置和启动**

如果是在一台机器上同时启动多个 RabbitMQ 节点来组建集群的话，只用上面介绍的方式启动第二、第三个节点将会因为节点名称和端口冲突导致启动失败。所以在每次调用 rabbitmq-server 命令前，设置环境变量 RABBITMQ\_NODENAME 和 RABBITMQ\_NODE\_PORT来明确指定唯一的节点名称和端口。

下面的例子端口号从5672开始，每个新启动的节点都加1，节点也分别命名为test\_rabbit\_1、test\_rabbit\_2、test\_rabbit\_3。

启动第1个节点：

RABBITMQ\_NODENAME=test\_rabbit\_1 RABBITMQ\_NODE\_PORT=5672 ./sbin/rabbitmq-server -detached

启动第2个节点：

RABBITMQ\_NODENAME=test\_rabbit\_2 RABBITMQ\_NODE\_PORT=5673 ./sbin/rabbitmq-server -detached

启动第2个节点前建议将 RabbitMQ 默认激活的插件关掉，否则会存在使用了某个插件的端口号冲突，导致节点启动不成功。

现在第2个节点和第1个节点都是独立节点，它们并不知道其他节点的存在。集群中除第一个节点外后加入的节点需要获取集群中的元数据，所以要先停止 Erlang 节点上运行的 RabbitMQ 应用程序，并重置该节点元数据，再加入并且获取集群的元数据，最后重新启动 RabbitMQ 应用程序。

停止第2个节点的应用程序：

./sbin/rabbitmqctl -n test\_rabbit\_2 stop\_app

重置第2个节点元数据：

./sbin/rabbitmqctl -n test\_rabbit\_2 reset

第2节点加入第1个节点组成的集群：

./sbin/rabbitmqctl -n test\_rabbit\_2 join\_cluster test\_rabbit\_1@localhost

启动第2个节点的应用程序

./sbin/rabbitmqctl -n test\_rabbit\_2 start\_app

第3个节点的配置过程和第2个节点类似：

RABBITMQ\_NODENAME=test\_rabbit\_3 RABBITMQ\_NODE\_PORT=5674 ./sbin/rabbitmq-server -detached

./sbin/rabbitmqctl -n test\_rabbit\_3 stop\_app

./sbin/rabbitmqctl -n test\_rabbit\_3 reset

./sbin/rabbitmqctl -n test\_rabbit\_3 join\_cluster test\_rabbit\_1@localhost

./sbin/rabbitmqctl -n test\_rabbit\_3 start\_app

RabbitMQ 集群运维

停止某个指定的节点，比如停止第2个节点：

RABBITMQ\_NODENAME=test\_rabbit\_2 ./sbin/rabbitmqctl stop

查看节点3的集群状态：

./sbin/rabbitmqctl -n test\_rabbit\_3 cluster\_status

## 四、MQTT

**1 MQTT（Message Queuing Telemetry Transport，消息队列遥测传输协议），**是一种基于发布/订阅（publish/subscribe）模式的**“轻量级”通讯协议**，该协议构建于TCP/IP协议上，由IBM在1999年发布。

**2 MQTT最大优点在于**，可以以极少的代码和有限的带宽，为连接远程设备提供实时可靠的消息服务。作为一种低开销、低带宽占用的即时通讯协议，使其在物联网、小型设备、移动应用等方面有较广泛的应用。

**3 MQTT是一个基于客户端-服务器的消息发布/订阅传输协议。**MQTT协议是轻量、简单、开放和易于实现的，这些特点使它适用范围非常广泛。在很多情况下，包括受限的环境中，如：机器与机器（M2M）通信和物联网（IoT）。其在，通过卫星链路通信传感器、偶尔拨号的医疗设备、智能家居、及一些小型化设备中已广泛使用。

**4 设计原则**

（1）精简，不添加可有可无的功能；

（2）发布/订阅（Pub/Sub）模式，方便消息在传感器之间传递；

（3）允许用户动态创建主题，零运维成本；

（4）把传输量降到最低以提高传输效率；

（5）把低带宽、高延迟、不稳定的网络等因素考虑在内；

（6）支持连续的会话控制；

（7）理解客户端计算能力可能很低；

（8）提供服务质量管理；

（9）假设数据不可知，不强求传输数据的类型与格式，保持灵活性。

**5 主要特性**

（1）使用发布/订阅消息模式，提供一对多的消息发布，解除应用程序耦合。MQTT的信息冗余远小于XMPP，,因为XMPP使用XML格式文本来传递数据。

（2）对负载内容屏蔽的消息传输。

（3）使用TCP/IP提供网络连接。主流的MQTT是基于TCP连接进行数据推送的，但是同样有基于UDP的版本，叫做MQTT-SN。这两种版本由于基于不同的连接方式，优缺点自然也就各有不同了。

（4）有三种消息发布服务质量：

@“至多一次”，环境传感器数据，丢失一次读记录无所谓，因为不久后还会有第二次发送。这一种方式主要普通APP的推送，倘若你的智能设备在消息推送时未联网，推送过去没收到，再次联网也就收不到了。

@“至少一次”，确保消息到达，但消息重复可能会发生。

@“只有一次”，确保消息到达一次。在一些要求比较严格的计费系统中，可以使用此级别。

（5）小型传输，开销很小（固定长度的头部是2字节），协议交换最小化，以降低网络流量。这就是为什么在介绍里说它非常适合“在物联网领域，传感器与服务器的通信，信息的收集”，要知道嵌入式设的运算能力和带宽都相对薄弱，使用这种协议来传递消息再适合不过了。

**6 MQTT协议实现方式**

实现MQTT协议需要客户端和服务器端通讯完成，在通讯过程中，MQTT协议中有三种身份：发布者（Publish）、代理（Broker）（服务器）、订阅者（Subscribe）。

其中，消息的发布者和订阅者都是客户端，消息代理是服务器，消息发布者可以同时是订阅者。

**7 MQTT传输的消息分为：主题（Topic）和负载（payload）两部分：**

**（1）Topic，**可以理解为消息的类型，订阅者订阅（Subscribe）后，就会收到该主题的消息内容（payload）；

**（2）payload，**可以理解为消息的内容，是指订阅者具体要使用的内容。

**8 MQTT客户端**

一个使用MQTT协议的应用程序或者设备，它总是建立到服务器的网络连接。客户端可以：

（1）发布其他客户端可能会订阅的信息；

（2）订阅其它客户端发布的消息；

（3）退订或删除应用程序的消息；

（4）断开与服务器连接。

**9 MQTT服务器**

MQTT服务器以称为“消息代理”（Broker），可以是一个应用程序或一台设备。它是位于消息发布者和订阅者之间，它可以：

（1）接受来自客户的网络连接；

（2）接受客户发布的应用信息；

（3）处理来自客户端的订阅和退订请求；

（4）向订阅的客户转发应用程序消息。

**订阅（Subscription）**

订阅包含主题筛选器（Topic Filter）和最大服务质量（QoS）。订阅会与一个会话（Session）关联。一个会话可以包含多个订阅。每一个会话中的每个订阅都有一个不同的主题筛选器。

**会话（Session）**

每个客户端与服务器建立连接后就是一个会话，客户端和服务器之间有状态交互。会话存在于一个网络之间，也可能在客户端和服务器之间跨越多个连续的网络连接。

**主题名（Topic Name**）

连接到一个应用程序消息的标签，该标签与服务器的订阅相匹配。服务器会将消息发送给订阅所匹配标签的每个客户端。

**主题筛选器（Topic Filter）**

一个对主题名通配符筛选器，在订阅表达式中使用，表示订阅所匹配到的多个主题。

**负载（Payload）**

消息订阅者所具体接收的内容。

**10 MQTT协议中的方法**

（1）Connect。等待与服务器建立连接。

（2）Disconnect。等待MQTT客户端完成所做的工作，并与服务器断开TCP/IP会话。

（3）Subscribe。等待完成订阅。

（4）UnSubscribe。等待服务器取消客户端的一个或多个topics订阅。

（5）Publish。MQTT客户端发送消息请求，发送完成后返回应用程序线程。

**1 1 Java实现**

**ServerMQTT类：**

import org.eclipse.paho.client.mqttv3.MqttClient;

import org.eclipse.paho.client.mqttv3.MqttConnectOptions;

import org.eclipse.paho.client.mqttv3.MqttDeliveryToken;

import org.eclipse.paho.client.mqttv3.MqttException;

import org.eclipse.paho.client.mqttv3.MqttMessage;

import org.eclipse.paho.client.mqttv3.MqttPersistenceException;

import org.eclipse.paho.client.mqttv3.MqttTopic;

import org.eclipse.paho.client.mqttv3.persist.MemoryPersistence;

public class ServerMQTT {

//tcp://MQTT安装的服务器地址:MQTT定义的端口号

public static final String HOST = "tcp://localhost:1883";

//定义一个主题

public static final String TOPIC = "topic11";

//定义MQTT的ID，可以在MQTT服务配置中指定

private static final String clientid = "server11";

private MqttClient client;

private MqttTopic topic11;

private String userName = "mosquitto";

private String passWord = "";

private MqttMessage message;

/\*\*

\* 构造函数

\* @throws MqttException

\*/

public ServerMQTT() throws MqttException {

// MemoryPersistence设置clientid的保存形式，默认为以内存保存

client = new MqttClient(HOST, clientid, new MemoryPersistence());

connect();

}

/\*\*

\* 用来连接服务器

\*/

private void connect() {

MqttConnectOptions options = new MqttConnectOptions();

options.setCleanSession(false);

options.setUserName(userName);

options.setPassword(passWord.toCharArray());

// 设置超时时间

options.setConnectionTimeout(10);

// 设置会话心跳时间

options.setKeepAliveInterval(20);

try {

client.setCallback(new PushCallback());

client.connect(options);

topic11 = client.getTopic(TOPIC);

} catch (Exception e) {

e.printStackTrace();

}

}

/\*\*

\*

\* @param topic

\* @param message

\* @throws MqttPersistenceException

\* @throws MqttException

\*/

public void publish(MqttTopic topic , MqttMessage message) throws MqttPersistenceException,

MqttException {

MqttDeliveryToken token = topic.publish(message);

token.waitForCompletion();

System.out.println("message is published completely! "

+ token.isComplete());

}

/\*\*

\* 启动入口

\* @param args

\* @throws MqttException

\*/

public static void main(String[] args) throws MqttException {

ServerMQTT server = new ServerMQTT();

server.message = new MqttMessage();

server.message.setQos(1);

server.message.setRetained(true);

server.message.setPayload("hello,topic11".getBytes());

server.publish(server.topic11 , server.message);

System.out.println(server.message.isRetained() + "------ratained状态");

}

}

**ClientMQTT类：**

import java.util.concurrent.ScheduledExecutorService;

import org.eclipse.paho.client.mqttv3.MqttClient;

import org.eclipse.paho.client.mqttv3.MqttConnectOptions;

import org.eclipse.paho.client.mqttv3.MqttException;

import org.eclipse.paho.client.mqttv3.MqttTopic;

import org.eclipse.paho.client.mqttv3.persist.MemoryPersistence;

public class ClientMQTT {

public static final String HOST = "tcp://localhost:1883";

public static final String TOPIC = "topic11";

private static final String clientid = "client11";

private MqttClient client;

private MqttConnectOptions options;

private String userName = "admin";

private String passWord = "password";

private ScheduledExecutorService scheduler;

private void start() {

try {

// host为主机名，clientid即连接MQTT的客户端ID，一般以唯一标识符表示，MemoryPersistence设置clientid的保存形式，默认为以内存保存

client = new MqttClient(HOST, clientid, new MemoryPersistence());

// MQTT的连接设置

options = new MqttConnectOptions();

// 设置是否清空session,这里如果设置为false表示服务器会保留客户端的连接记录，这里设置为true表示每次连接到服务器都以新的身份连接

options.setCleanSession(true);

// 设置连接的用户名

options.setUserName(userName);

// 设置连接的密码

options.setPassword(passWord.toCharArray());

// 设置超时时间 单位为秒

options.setConnectionTimeout(10);

// 设置会话心跳时间 单位为秒 服务器会每隔1.5\*20秒的时间向客户端发送个消息判断客户端是否在线，但这个方法并没有重连的机制

options.setKeepAliveInterval(20);

// 设置回调

client.setCallback(new PushCallback());

MqttTopic topic = client.getTopic(TOPIC);

//setWill方法，如果项目中需要知道客户端是否掉线可以调用该方法。设置最终端口的通知消息

options.setWill(topic, "close".getBytes(), 2, true);

client.connect(options);

//订阅消息

int[] Qos = {1};

String[] topic1 = {TOPIC};

client.subscribe(topic1, Qos);

} catch (Exception e) {

e.printStackTrace();

}

}

public static void main(String[] args) throws MqttException {

ClientMQTT client = new ClientMQTT();

client.start();

}

}

**PushCallback类：**

import org.eclipse.paho.client.mqttv3.IMqttDeliveryToken;

import org.eclipse.paho.client.mqttv3.MqttCallback;

import org.eclipse.paho.client.mqttv3.MqttMessage;

/\*\*

\* 发布消息的回调类

\*

\* 必须实现MqttCallback的接口并实现对应的相关接口方法CallBack 类将实现 MqttCallBack。

\* 每个客户机标识都需要一个回调实例。在此示例中，构造函数传递客户机标识以另存为实例数据。

\* 在回调中，将它用来标识已经启动了该回调的哪个实例。

\* 必须在回调类中实现三个方法：

\*

\* public void messageArrived(MqttTopic topic, MqttMessage message)接收已经预订的发布。

\*

\* public void connectionLost(Throwable cause)在断开连接时调用。

\*

\* public void deliveryComplete(MqttDeliveryToken token))

\* 接收到已经发布的 QoS 1 或 QoS 2 消息的传递令牌时调用。

\* 由 MqttClient.connect 激活此回调。

\*

\*/

public class PushCallback implements MqttCallback {

public void connectionLost(Throwable cause) {

// 连接丢失后，一般在这里面进行重连

System.out.println("连接断开，可以做重连");

}

public void deliveryComplete(IMqttDeliveryToken token) {

System.out.println("deliveryComplete---------" + token.isComplete());

}

public void messageArrived(String topic, MqttMessage message) throws Exception {

// subscribe后得到的消息会执行到这里面

System.out.println("接收消息主题 : " + topic);

System.out.println("接收消息Qos : " + message.getQos());

System.out.println("接收消息内容 : " + new String(message.getPayload()));

}

}

**server类启动的结果：**

deliveryComplete---------true

message is published completely! true

true------ratained状态

**client类启动的结果：**

接收消息主题 : topic11

接收消息Qos : 1

接收消息内容 : hello,topic11

## ZeroMq

## 1  ZeroMQ概述

ZeroMQ是一种基于**消息队列的多线程网络库，其对套接字类型、连接处理、帧、甚至路由的底层细节进行抽象，提供跨越多种传输协议的套接字**。ZeroMQ是网络通信中新的一层，介于应用层和传输层之间（按照TCP/IP划分），其是一个可伸缩层，可并行运行，分散在分布式系统间。

2  系统架构

2.1总体架构

ZeroMQ几乎所有的I/O操作都是异步的，主线程不会被阻塞。ZeroMQ会根据用户调用zmq\_init函数时传入的接口参数，创建对应数量的I/O Thread。每个I/O Thread都有与之绑定的Poller，Poller采用经典的Reactor模式实现，Poller根据不同操作系统平台使用不同的网络I/O模型（select、poll、epoll、devpoll、kequeue等）。主线程与I/O线程通过Mail Box传递消息来进行通信。Server开始监听或者Client发起连接时，在主线程中创建zmq\_connecter或zmq\_listener，通过Mail Box发消息的形式将其绑定到I/O线程，I/O线程会把zmq\_connecter或zmq\_listener添加到Poller中用以侦听读/写事件。Server与Client在第一次通信时，会创建zmq\_init来发送identity，用以进行认证。认证结束后，双方会为此次连接创建Session，以后双方就通过Session进行通信。每个Session都会关联到相应的读/写管道， 主线程收发消息只是分别从管道中读/写数据。Session并不实际跟kernel交换I/O数据，而是通过plugin到Session中的Engine来与kernel交换I/O数据。

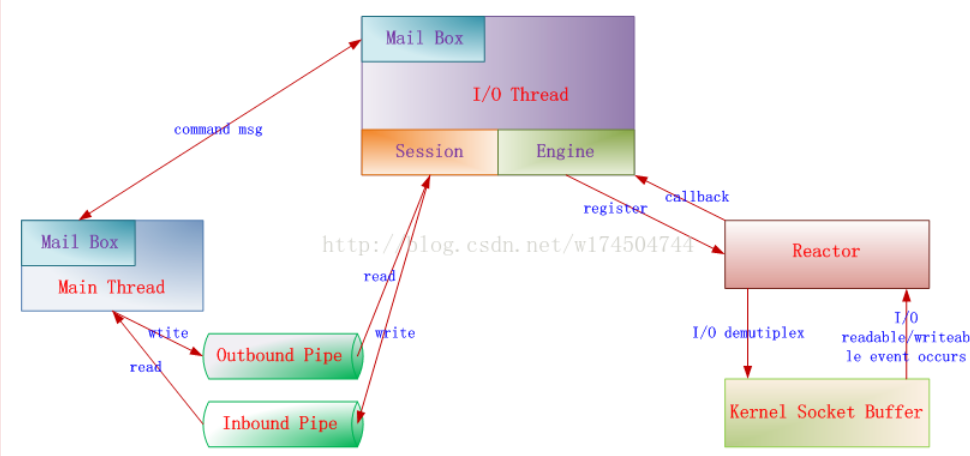


图1总体架构

2.2所处层次

         ZeroMQ不是单独的服务或者程序，仅仅是一套组件，其封装了网络通信、消息队列、线程调度等功能，向上层提供简洁的API，应用程序通过加载库文件，调用API函数来实现高性能网络通信。



图2所处层次

2.3消息模型

ZeroMQ将消息通信分成4种模型，分别是一对一结对模型（Exclusive-Pair）、请求回应模型（Request-Reply）、发布订阅模型（Publish-Subscribe）、推拉模型（Push-Pull）。这4种模型总结出了通用的网络通信模型，在实际中可以根据应用需要，组合其中的2种或多种模型来形成自己的解决方案。

2.3.1   一对一结对模型

         最简单的1:1消息通信模型，可以认为是一个TCP Connection，但是TCP Server只能接受一个连接。数据可以双向流动，这点不同于后面的请求回应模型。

2.3.2   请求回应模型

由请求端发起请求，然后等待回应端应答。一个请求必须对应一个回应，从请求端的角度来看是发-收配对，从回应端的角度是收-发对。跟一对一结对模型的区别在于请求端可以是1~N个。该模型主要用于远程调用及任务分配等。Echo服务就是这种经典模型的应用。

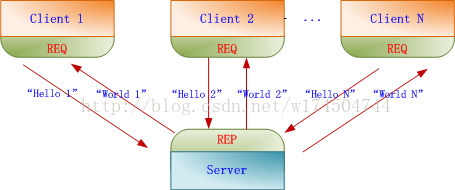


图3请求回应模型

2.3.3   发布订阅模型

         发布端单向分发数据，且不关心是否把全部信息发送给订阅端。如果发布端开始发布信息时，订阅端尚未连接上来，则这些信息会被直接丢弃。订阅端未连接导致信息丢失的问题，可以通过与请求回应模型组合来解决。订阅端只负责接收，而不能反馈，且在订阅端消费速度慢于发布端的情况下，会在订阅端堆积数据。该模型主要用于数据分发。天气预报、微博明星粉丝可以应用这种经典模型。

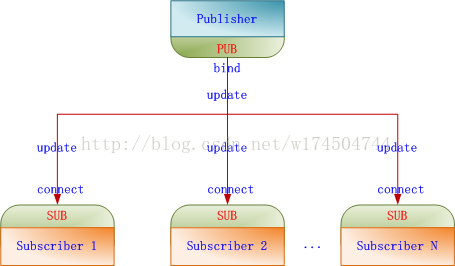


图4发布订阅模型

2.3.4   推拉模型

Server端作为Push端，而Client端作为Pull端，如果有多个Client端同时连接到Server端，则Server端会在内部做一个负载均衡，采用平均分配的算法，将所有消息均衡发布到Client端上。与发布订阅模型相比，推拉模型在没有消费者的情况下，发布的消息不会被消耗掉；在消费者能力不够的情况下，能够提供多消费者并行消费解决方案。该模型主要用于多任务并行。

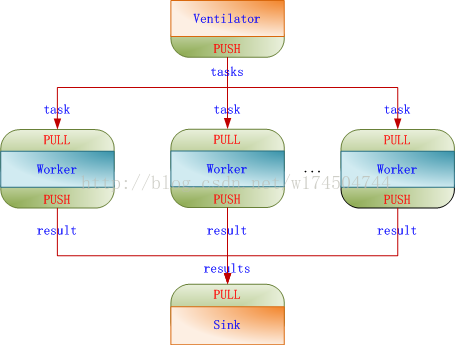


图5 推拉模型

2.4通信协议

提供进程内、进程间、机器间、广播等四种通信协议。通信协议配置简单，用类似于URL形式的字符串指定即可，格式分别为inproc://、ipc://、tcp://、pgm://。ZeroMQ会自动根据指定的字符串解析出协议、地址、端口号等信息。

4  性能分析

目前，市面上类似的产品不少，主要有4种：MSMQ（微软产品）、ActiveMQ（Java）、RabbitMQ(Erlang)、ZeroMQ（C++）。除ZeroMQ外，其它3款产品都是一个单独服务或者进程，需要单独安装和运行，且对环境有一定依赖。其中，MSMQ在非Windows平台下安装非常复杂，ActiveMQ需要目标机器上已经安装了Java，RabbitMQ需要Erlang环境。而ZeroMQ是以库的形式存在，由应用程序加载、运行即可。但是ZeroMQ仅提供非持久性的消息队列。

图7是来自于Internet的性能测试数据。显示的是每秒钟发送和接受的消息数。整个过程共产生1百万条1K的消息，测试环境为Windows Vista。从测试数据可以看出，ZeroMQ的性能远远高于其它3个MQ。

但是测试数据仅供参考，因为缺少必须的环境参数和性能指标，比如：CPU参数、内存参数、消息模型、通信协议、极限时消耗CPU百分比、极限时消耗内存百分比等。

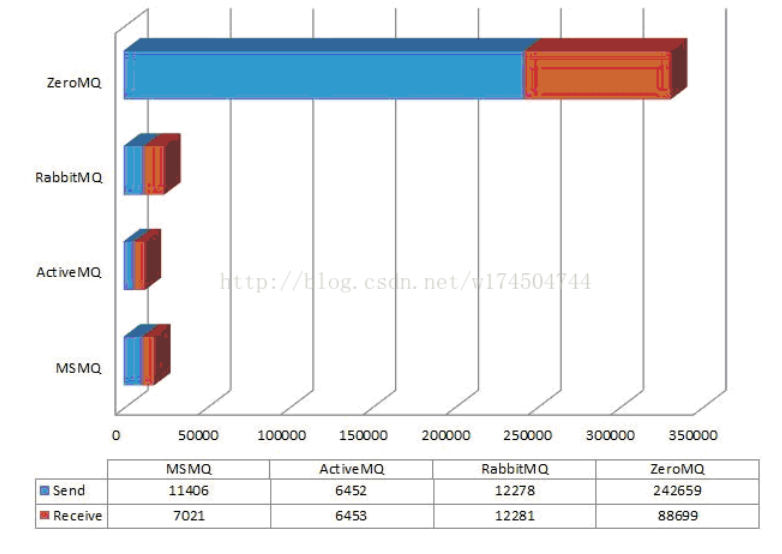
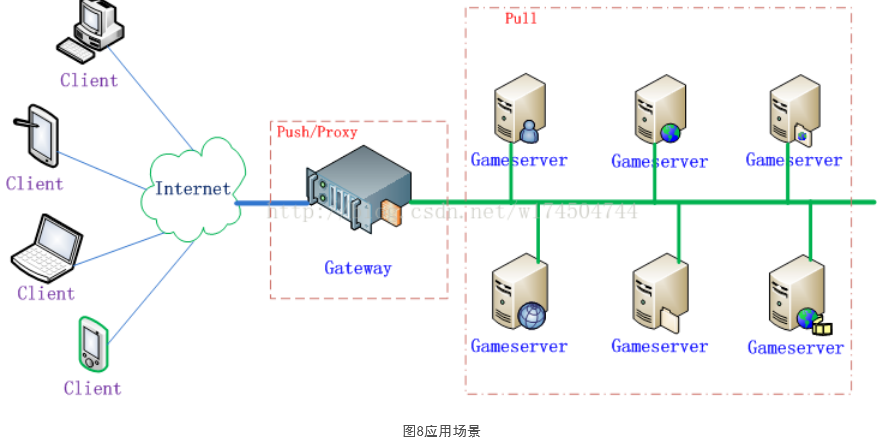


图7性能测试

5  应用场景

应用ZeroMQ的Push-Pull模型实现联众游戏服务器的“热插拔”、负载均衡和消息派发。按照如图8部署服务器，Push端充当Gateway，作为一组游戏服务器集群最上层的一个Proxy，起负载均衡的作用，所有Gameserver作为Pull端。当一个请求到达Push端（Gateway）时，Push端根据一定的分配策略将任务派发到Pull端（Gameserver）。以联众某款游戏A为例，游戏A刚上线时，预计最大同时在线人数是10W，单台Gameserver并发处理能力为1W，需要10台Gameserver，由于游戏A可玩性非常好，半个月后最大同时在线人数暴增到50W，那么不需要在某天的凌晨将Gateway和Gameserver停机，只需要随时在机房新添加40台Gameserver，启动并连接到Gateway即可。

ZeroMQ中对Client和Server的启动顺序没有要求，Gameserver之间如果需要通信的话，Gameserver的应用层不需要管理这些细节，ZeroMQ已经做了重连处理。



6  总结

6.1简单

1、仅仅提供24个API接口，风格类似于BSD Socket。

2、处理了网络异常，包括连接异常中断、重连等。

3、改变TCP基于字节流收发数据的方式，处理了粘包、半包等问题，以msg为单位收发数据，结合Protocol Buffers，可以对应用层彻底屏蔽网络通信层。

4、对大数据通过SENDMORE/RECVMORE提供分包收发机制。

5、通过线程间数据流动来保证同一时刻任何数据都只会被一个线程持有，以此实现多线程的“去锁化”。

6、通过高水位HWM来控制流量，用交换SWAP来转储内存数据，弥补HWM丢失数据的缺陷。

7、服务器端和客户端的启动没有先后顺序。

6.2灵活

1、支持多种通信协议，可以灵活地适应多种通信环境，包括进程内、进程间、机器间、广播。

2、支持多种消息模型，消息模型之间可以相互组合，形成特定的解决方案。

6.3跨平台

支持Linux、Windows、OS X等。

6.4多语言

可以绑定C、C++、Java、.NET、Python等30多种开发语言。

6.5高性能

相对同类产品，性能卓越。