### 一、消息队列使用场景或者其好处

消息队列一般是在项目中，将一些无需即时返回且耗时的操作提取出来，进行了异步处理，而这种异步处理的方式大大的节省了服务器的请求响应时间，从而提高了系统的吞吐量。

在项目启动之初来预测将来项目会碰到什么需求，是极其困难的。消息队列在处理过程中间插入了一个隐含的、基于数据的接口层，两边的处理过程都要实现这一接口。这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。消息队列可以解决这样一个问题，也就是其解耦性。解耦伴随的好处就是降低冗余，灵活，易于扩展。

峰值处理能力：当你的应用上了Hacker News的首页，你将发现访问流量攀升到一个不同寻常的水平。在访问量剧增的情况下，你的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增长的访问压力，而不是因为超出负荷的请求而完全崩溃。   
消息队列还有可恢复性、异步通信、缓冲………等各种好处，在此不一一介绍，用到自然理解。

### 二、RabbitMQ来源

RabbitMQ是用**Erlang**实现的一个高并发高可靠**AMQP**消息队列服务器。

显然，RabbitMQ跟Erlang和AMQP有关。下面简单介绍一下Erlang和AMQP。

Erlang是一门动态类型的函数式编程语言，它也是一门解释型语言，由Erlang虚拟机解释执行。从语言模型上说，Erlang是基于Actor模型的实现。在Actor模型里面，万物皆Actor，每个Actor都封装着内部状态，Actor相互之间只能通过消息传递这一种方式来进行通信。对应到Erlang里，每个Actor对应着一个Erlang进程，进程之间通过消息传递进行通信。相比共享内存，进程间通过消息传递来通信带来的直接好处就是消除了直接的锁开销(不考虑Erlang虚拟机底层实现中的锁应用)。

AMQP(Advanced Message Queue Protocol)定义了一种消息系统规范。这个规范描述了在一个分布式的系统中各个子系统如何通过消息交互。而RabbitMQ则是AMQP的一种基于erlang的实现。AMQP将分布式系统中各个子系统隔离开来，子系统之间不再有依赖。子系统仅依赖于消息。子系统不关心消息的发送者，也不关心消息的接受者。

这里不必要对Erlang和AMQP作过于深入介绍，毕竟本文RabbitMQ才是主角哦，哈哈。下面直接看主角表演（实例）啦，至于主角的一些不得不深入介绍的点我们放到最后面。

### 三、RabbitMQ实例（Java）

**3.1、环境配置**

RabbitMQ的运行需要erlang的支持，因此我们先安装erlang。   
32位下载地址：<http://www.erlang.org/download/otp_win64_18.2.1.exe>   
64位下载地址：<http://www.erlang.org/download/otp_win32_18.2.1.exe>   
双击选择默认安装就好。

前面我们也讲到RabbitMQ就是一个服务器，下面我们就安装对应服务器。   
下载地址：<http://www.rabbitmq.com/releases/rabbitmq-server/v3.3.4/rabbitmq-server-3.3.4.exe>   
双击选择默认安装就好，安装好之后需要启动服务，cmd，进入到安装目录的sbin文件夹下，命令如下：

cd C:\Program Files (x86)\RabbitMQ Server\rabbitmq\_server-3.3.4\sbin

rabbitmq-server start



博主的之前启动过了，所以报错，如果你的也启动了就没问题了。

接下来自然是jar包依赖，本文工程采用eclipse + maven，maven依赖如下：

<!-- rabbitmq相关依赖 -->

<dependency>

<groupId>com.rabbitmq</groupId>

<artifactId>amqp-client</artifactId>

<version>3.0.4</version>

</dependency>

<!-- 序列化相关依赖 -->

<dependency>

<groupId>commons-lang</groupId>

<artifactId>commons-lang</artifactId>

<version>2.6</version>

</dependency>

因为后续例子里面有用到序列化的，因此加上序列化工具包相关依赖。

**3.2、例子一代码和效果**

新建发送者Send.java，代码如下：

package com.luo.rabbit.test.one;

import com.rabbitmq.client.Channel;

import com.rabbitmq.client.Connection;

import com.rabbitmq.client.ConnectionFactory;

public class Send {

//队列名称

private final static String QUEUE\_NAME = "queue";

public static void main(String[] argv) throws java.io.IOException

{

/\*\*

\* 创建连接连接到MabbitMQ

\*/

ConnectionFactory factory = new ConnectionFactory();

//设置MabbitMQ所在主机ip或者主机名

factory.setHost("127.0.0.1");

//创建一个连接

Connection connection = factory.newConnection();

//创建一个频道

Channel channel = connection.createChannel();

//指定一个队列

channel.queueDeclare(QUEUE\_NAME, false, false, false, null);

//发送的消息

String message = "hello world!";

//往队列中发出一条消息

channel.basicPublish("", QUEUE\_NAME, null, message.getBytes());

System.out.println("Sent '" + message + "'");

//关闭频道和连接

channel.close();

connection.close();

}

}

新建接收者Recv.java，代码如下：

package com.luo.rabbit.test.one;

import com.rabbitmq.client.Channel;

import com.rabbitmq.client.Connection;

import com.rabbitmq.client.ConnectionFactory;

import com.rabbitmq.client.QueueingConsumer;

public class Recv {

//队列名称

private final static String QUEUE\_NAME = "queue";

public static void main(String[] argv) throws java.io.IOException,

java.lang.InterruptedException

{

//打开连接和创建频道，与发送端一样

ConnectionFactory factory = new ConnectionFactory();

//设置MabbitMQ所在主机ip或者主机名

factory.setHost("127.0.0.1");

Connection connection = factory.newConnection();

Channel channel = connection.createChannel();

//声明队列，主要为了防止消息接收者先运行此程序，队列还不存在时创建队列。

channel.queueDeclare(QUEUE\_NAME, false, false, false, null);

System.out.println("Waiting for messages. To exit press CTRL+C");

//创建队列消费者

QueueingConsumer consumer = new QueueingConsumer(channel);

//指定消费队列

channel.basicConsume(QUEUE\_NAME, true, consumer);

while (true)

{

//nextDelivery是一个阻塞方法（内部实现其实是阻塞队列的take方法）

QueueingConsumer.Delivery delivery = consumer.nextDelivery();

String message = new String(delivery.getBody());

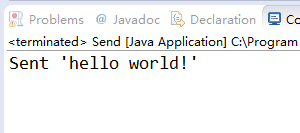
System.out.println("Received '" + message + "'");

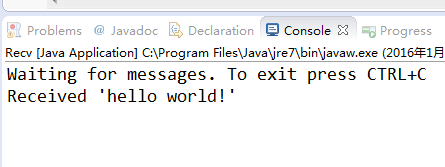
}

}

}

分别运行这两个类，先后顺序没有关系，先运行发送者再运行接收者，效果如下：





**3.3、例子二代码和效果**

例子一可能通俗易懂，但是并不是很规范，而且有些缺陷，比如我要发送一个对象过去呢？下面看另外一个例子：

首先建一个连接类，因为发送者和接收者的连接代码都是一样的，之后让二者继承这个连接类即可。连接类代码BaseConnector.java：

package com.luo.rabbit.test.two;

import java.io.IOException;

import com.rabbitmq.client.Channel;

import com.rabbitmq.client.Connection;

import com.rabbitmq.client.ConnectionFactory;

public class BaseConnector {

protected Channel channel;

protected Connection connection;

protected String queueName;

public BaseConnector(String queueName) throws IOException{

this.queueName = queueName;

//打开连接和创建频道

ConnectionFactory factory = new ConnectionFactory();

//设置MabbitMQ所在主机ip或者主机名 127.0.0.1即localhost

factory.setHost("127.0.0.1");

//创建连接

connection = factory.newConnection();

//创建频道

channel = connection.createChannel();

//声明创建队列

channel.queueDeclare(queueName, false, false, false, null);

}

}

发送者Sender.java：

package com.luo.rabbit.test.two;

import java.io.IOException;

import java.io.Serializable;

import org.apache.commons.lang.SerializationUtils;

public class Sender extends BaseConnector {

public Sender(String queueName) throws IOException {

super(queueName);

}

public void sendMessage(Serializable object) throws IOException {

channel.basicPublish("",queueName, null, SerializationUtils.serialize(object));

}

}

前面讲过，我们想发送一个对象给接受者，因此，我们先新建一个对象，因为发送过程需要序列化，因此这里需要实现java.io.Serializable接口：

package com.luo.rabbit.test.two;

import java.io.Serializable;

public class MessageInfo implements Serializable {

private static final long serialVersionUID = 1L;

//渠道

private String channel;

//来源

private String content;

public String getChannel() {

return channel;

}

public void setChannel(String channel) {

this.channel = channel;

}

public String getContent() {

return content;

}

public void setContent(String content) {

this.content = content;

}

}

关于序列化，这里小宝鸽就再唠叨两句，序列化就是将一个对象的状态（各个属性量）保存起来，然后在适当的时候再获得。序列化分为两大部分：序列化和反序列化。序列化是这个过程的第一部分，将数据分解成字节流，以便存储在文件中或在网络上传输。反序列化就是打开字节流并重构对象。对象序列化不仅要将基本数据类型转换成字节表示，有时还要恢复数据。恢复数据要求有恢复数据的对象实例。

接收者代码Receiver.java：

package com.luo.rabbit.test.two;

import java.io.IOException;

import org.apache.commons.lang.SerializationUtils;

import com.rabbitmq.client.AMQP.BasicProperties;

import com.rabbitmq.client.Envelope;

import com.rabbitmq.client.Consumer;

import com.rabbitmq.client.ShutdownSignalException;

public class Receiver extends BaseConnector implements Runnable, Consumer {

public Receiver(String queueName) throws IOException {

super(queueName);

}

//实现Runnable的run方法

public void run() {

try {

channel.basicConsume(queueName, true,this);

} catch (IOException e) {

e.printStackTrace();

}

}

/\*\*

\* 下面这些方法都是实现Consumer接口的

\*\*/

//当消费者注册完成自动调用

public void handleConsumeOk(String consumerTag) {

System.out.println("Consumer "+consumerTag +" registered");

}

//当消费者接收到消息会自动调用

public void handleDelivery(String consumerTag, Envelope env,

BasicProperties props, byte[] body) throws IOException {

MessageInfo messageInfo = (MessageInfo)SerializationUtils.deserialize(body);

System.out.println("Message ( "

+ "channel : " + messageInfo.getChannel()

+ " , content : " + messageInfo.getContent()

+ " ) received.");

}

//下面这些方法可以暂时不用理会

public void handleCancelOk(String consumerTag) {

}

public void handleCancel(String consumerTag) throws IOException {

}

public void handleShutdownSignal(String consumerTag,

ShutdownSignalException sig) {

}

public void handleRecoverOk(String consumerTag) {

}

}

这里，接收者实现了，Runnable接口和com.rabbitmq.client.Consumer接口。

实现Runnable接口的目的是为了实现多线程，java实现多线程的方式有两种：一种是继承Thread类，一种是实现Runnable接口。详情请看这篇文章：<http://developer.51cto.com/art/201203/321042.htm>。

实现Consumer接口的目的是什么呢？猿友们应有看到实例一中的接收者代码：

//指定消费队列

channel.basicConsume(QUEUE\_NAME, true, consumer);

最后一个参数是需要传递com.rabbitmq.client.Consumer参数的，实现了Consumer接口之后我们只需要传递this就好了。另外，Consumer有很多方法，上面代码除了构造方法和run方法（run是实现Runnable接口的），其他都是实现Consumer接口的，这些方法的具体含义，大家可以直接看com.rabbitmq.client.Consumer源码。

接下来就是测试类了Test.java：

package com.luo.rabbit.test.two;

public class Test {

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

Receiver receiver = new Receiver("testQueue");

Thread receiverThread = new Thread(receiver);

receiverThread.start();

Sender sender = new Sender("testQueue");

for (int i = 0; i < 5; i++) {

MessageInfo messageInfo = new MessageInfo();

messageInfo.setChannel("test");

messageInfo.setContent("msg" + i);

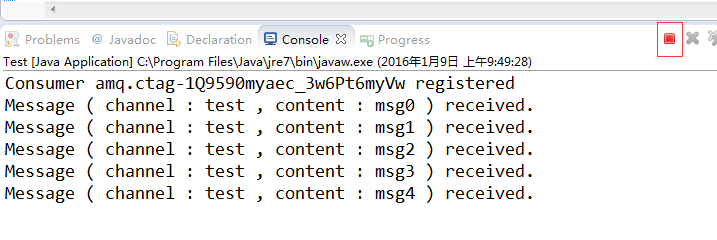
sender.sendMessage(messageInfo);

}

}

}

运行效果：



记得运行完成之后一定要把进程关掉，不然你每运行一次Test.java就会开启一个进程，之后会出现什么问题呢？我是十分建议大家试试，会有惊喜哦，哈哈，惊喜就是，发送的消息会平均（数量平均）的出现到各个接收者的控制台。不妨将发送的数量改大一点试试。

### 四、RabbitMQ使用的道具的具体介绍

RabbitMQ是用**Erlang**实现的一个高并发高可靠**AMQP**消息队列服务器。

Erlang就是RabbitMQ的一个依赖环境，这里没什么好说的。我们更加关注它的一身表演技巧哪里来的，这里就看AMQP吧，看完AMQP之后估计你会对RabbitMQ的理解更加深刻。

**开始吧**   
AMQP当中有四个概念非常重要：虚拟主机（virtual host），交换机（exchange），队列（queue）和绑定（binding）。一个虚拟主机持有一组交换机、队列和绑定。为什么需要多个虚拟主机呢？很简单，RabbitMQ当中，用户只能在虚拟主机的粒度进行权限控制。因此，如果需要禁止A组访问B组的交换机/队列/绑定，必须为A和B分别创建一个虚拟主机。每一个RabbitMQ服务器都有一个默认的虚拟主机“/”。如果这就够了，那现在就可以开始了。

**交换机，队列，还有绑定……天哪！**   
刚开始我思维的列车就是在这里脱轨的…… 这些鬼东西怎么结合起来的？

队列（Queues）是你的消息（messages）的终点，可以理解成装消息的容器。消息就一直在里面，直到有客户端（也就是消费者，Consumer）连接到这个队列并且将其取走为止。不过。你可以将一个队列配置成这样的：一旦消息进入这个队列，biu~，它就烟消云散了。这个有点跑题了……

需要记住的是，队列是由消费者（Consumer）通过程序建立的，不是通过配置文件或者命令行工具。这没什么问题，如果一个消费者试图创建一个已经存在的队列，RabbitMQ就会起来拍拍他的脑袋，笑一笑，然后忽略这个请求。因此你可以将消息队列的配置写在应用程序的代码里面。这个概念不错。

OK，你已经创建并且连接到了你的队列，你的消费者程序正在百无聊赖的敲着手指等待消息的到来，敲啊，敲啊…… 没有消息。发生了什么？你当然需要先把一个消息放进队列才行。不过要做这个，你需要一个交换机（Exchange）……

交换机可以理解成具有路由表的路由程序，仅此而已。每个消息都有一个称为路由键（routing key）的属性，就是一个简单的字符串。交换机当中有一系列的绑定（binding），即路由规则（routes），例如，指明具有路由键 “X” 的消息要到名为timbuku的队列当中去。先不讨论这个，我们有点超前了。

你的消费者程序要负责创建你的交换机们（复数）。啥？你是说你可以有多个交换机？是的，这个可以有，不过为啥？很简单，每个交换机在自己独立的进程当中执行，因此增加多个交换机就是增加多个进程，可以充分利用服务器上的CPU核以便达到更高的效率。例如，在一个8核的服务器上，可以创建5个交换机来用5个核，另外3个核留下来做消息处理。类似的，在RabbitMQ的集群当中，你可以用类似的思路来扩展交换机一边获取更高的吞吐量。

OK，你已经创建了一个交换机。但是他并不知道要把消息送到哪个队列。你需要路由规则，即绑定（binding）。一个绑定就是一个类似这样的规则：将交换机“desert（沙漠）”当中具有路由键“阿里巴巴”的消息送到队列“hideout（山洞）”里面去。换句话说，一个绑定就是一个基于路由键将交换机和队列连接起来的路由规则。例如，具有路由键“audit”的消息需要被送到两个队列，“log-forever”和“alert-the-big-dude”。要做到这个，就需要创建两个绑定，每个都连接一个交换机和一个队列，两者都是由“audit”路由键触发。在这种情况下，交换机会复制一份消息并且把它们分别发送到两个队列当中。交换机不过就是一个由绑定构成的路由表。

现在复杂的东西来了：交换机有多种类型。他们都是做路由的，不过接受不同类型的绑定。为什么不创建一种交换机来处理所有类型的路由规则呢？因为每种规则用来做匹配分子的CPU开销是不同的。例如，一个“topic”类型的交换机试图将消息的路由键与类似“dogs.\*”的模式进行匹配。匹配这种末端的通配符比直接将路由键与“dogs”比较（“direct”类型的交换机）要消耗更多的CPU。如果你不需要“topic”类型的交换机带来的灵活性，你可以通过使用“direct”类型的交换机获取更高的处理效率。那么有哪些类型，他们又是怎么处理的呢？

上面这些都是参考另外一篇文章的，<http://blog.ftofficer.com/2010/03/translation-rabbitmq-python-rabbits-and-warrens/>，当然这篇文章的实例是Python的，但是我们不看他的实例，只看他吹水的那部分，哈哈。

### 五、源码工程下载

<http://download.csdn.net/detail/u013142781/9396830>

小宝鸽向来有个坏习惯，即便博客里面已经将全部代码贴出来了，还是会提供源码工程供大家下载，哈哈。

有些时候有些猿友经常会问，写一篇博客很花时间吧，我不能假装跟你说不花时间。虽然花时间，但是当你看到方向，看到了目标，可以将自己学习的东西分享出来，你就会很有动力了，根本停不下来。