

- RabbitMQ 使用场景
 - 服务解耦
 - 流量削峰
 - 异步调用
- rabbitmq 基本概念
 - Exchange
 - Message Queue
 - Binding Key
 - Routing Key
- rabbitmq安装
 - 安装erlang语言库
 - rabbitmq官方精简的Erlang语言包
 - 下载和安装
 - 安装socat依赖
 - socat依赖包
 - 下载和安装
 - 安装rabbitmq
 - rabbitmq安装包
 - 下载和安装
 - rabbitmq启动和停止命令
 - rabbitmq管理界面
 - 启用管理界面
 - 访问
 - 添加用户
 - 添加用户
 - 设置访问权限
 - 开放客户端连接端口
- rabbitmq六种工作模式
 - 简单模式
 - pom.xml
 - 生产者发送消息
 - 消费者接收消息
 - 工作模式
 - 生产者发送消息
 - 消费者接收消息
 - 运行测试
 - 消息确认
 - 消息持久化
 - 合理地分发
 - 生产者代码

- 消费者代码
- 发布订阅模式
 - Exchanges 交换机
 - 绑定 Bindings
 - 完成的代码
 - 生产者
 - 消费者
- 路由模式
 - 绑定 Bindings
 - 直连交换机 Direct exchange
 - 多重绑定 Multiple bindings
 - 发送日志
 - 订阅
 - 完整的代码
 - 生产者
 - 消费者
- 主题模式
 - 主题交换机 Topic exchange
 - 完成的代码
 - 生产者
 - 消费者
- RPC模式
 - 客户端
 - 回调队列 Callback Queue
 - 关联id (correlationId):
 - 小结
 - 完成的代码
 - 服务器端
 - 客户端
- virtual host
 - 创建virtual host: `/pd`
 - 设置虚拟机的用户访问权限
- 拼多多商城整合 rabbitmq
 - 订单存储的解耦
 - 生产者-发送订单
 - pom.xml 添加依赖
 - application.yml
 - 修改主程序 RunPdAPP
 - 修改 OrderServiceImpl
 - 消费者-接收订单,并保存到数据库
 - pd-web项目复制为pd-order-consumer
 - 修改 application.yml

- 删除无关代码
- 新建 OrderConsumer
- 修改 OrderServiceImpl 的 saveOrder() 方法
- 手动确认
 - application.yml
 - OrderConsumer

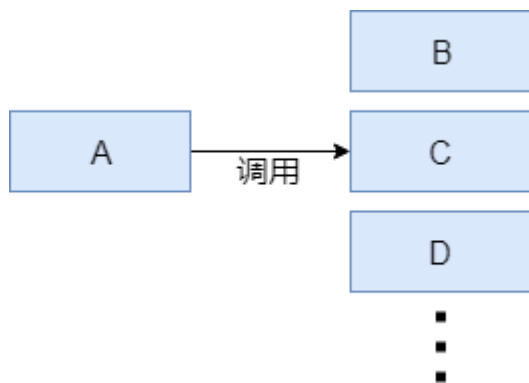
RabbitMQ 使用场景

服务解耦

假设有这样一个场景, 服务A产生数据, 而服务B,C,D需要这些数据, 那么我们可以在A服务中直接调用B,C,D服务,把数据传递到下游服务即可

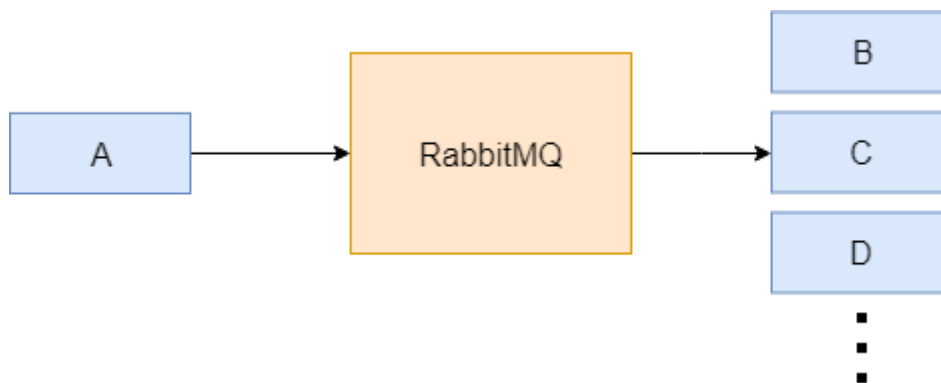
但是,随着我们的应用规模不断扩大,会有更多的服务需要A的数据,如果有几十甚至几百个下游服务,而且会不断变更,再加上还要考虑下游服务出错的情况,那么A服务中调用代码的维护会极为困难

这是由于服务之间耦合度过于紧密



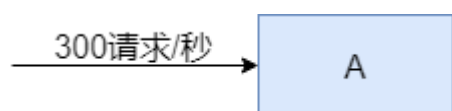
再来考虑用RabbitMQ解耦的情况

A服务只需要向消息服务器发送消息,而不用考虑谁需要这些数据;下游服务如果需要数据,自行从消息服务器订阅消息,不再需要数据时则取消订阅即可



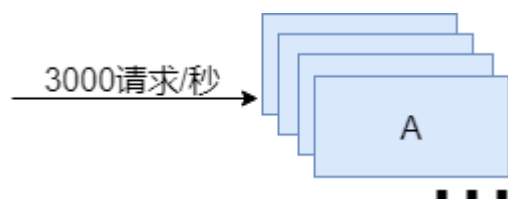
流量削峰

假设我们有一个应用,平时访问量是每秒300请求,我们用一台服务器即可轻松应对



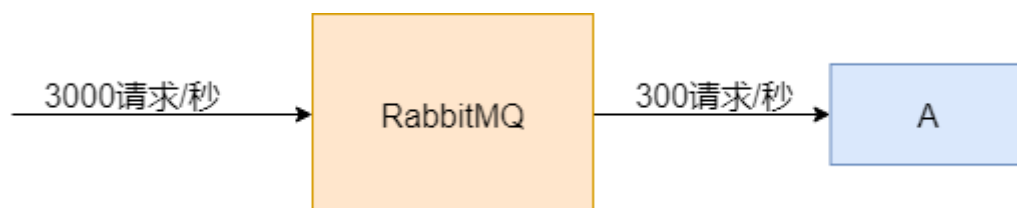
而在高峰期,访问量瞬间翻了十倍,达到每秒3000次请求,那么单台服务器肯定无法应对,这时我们可以考虑增加到10台服务器,来分散访问压力

但如果这种瞬时高峰的情况每天只出现一次,每次只有半小时,那么我们10台服务器在多数时间都只分担每秒几十次请求,这样就有点浪费资源了



这种情况,我们就可以使用RabbitMQ来进行流量削峰,高峰情况下,瞬间出现的大量请求数据,先发送到消息队列服务器,排队等待被处理,而我们的应用,可以慢慢的从消息队列接收请求数据进行处理,这样把数据处理时间拉长,以减轻瞬时压力

这是消息队列服务器非常典型的应用场景

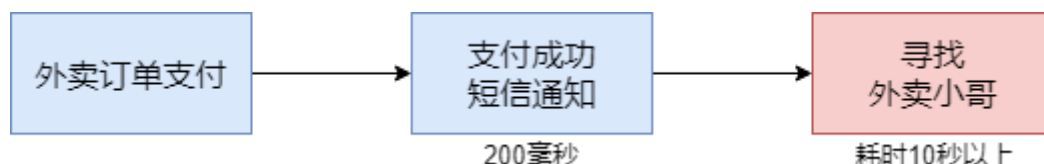


异步调用

考虑定外卖支付成功的情况

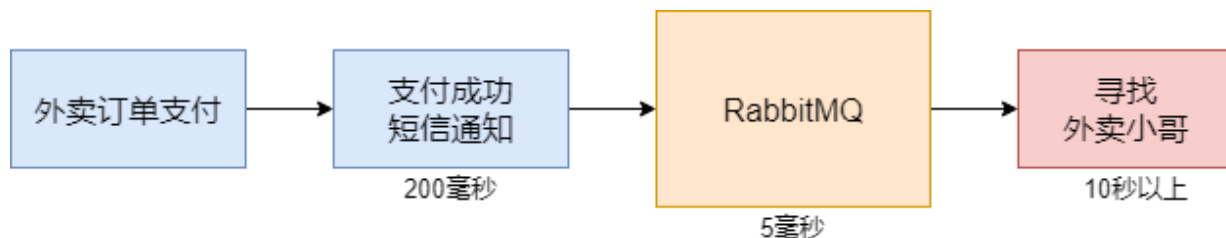
支付后要发送支付成功的通知,再寻找外卖小哥来进行配送,而寻找外卖小哥的过程非常耗时,尤其是高峰期,可能要等待几十秒甚至更长

这样就造成整条调用链路响应非常缓慢



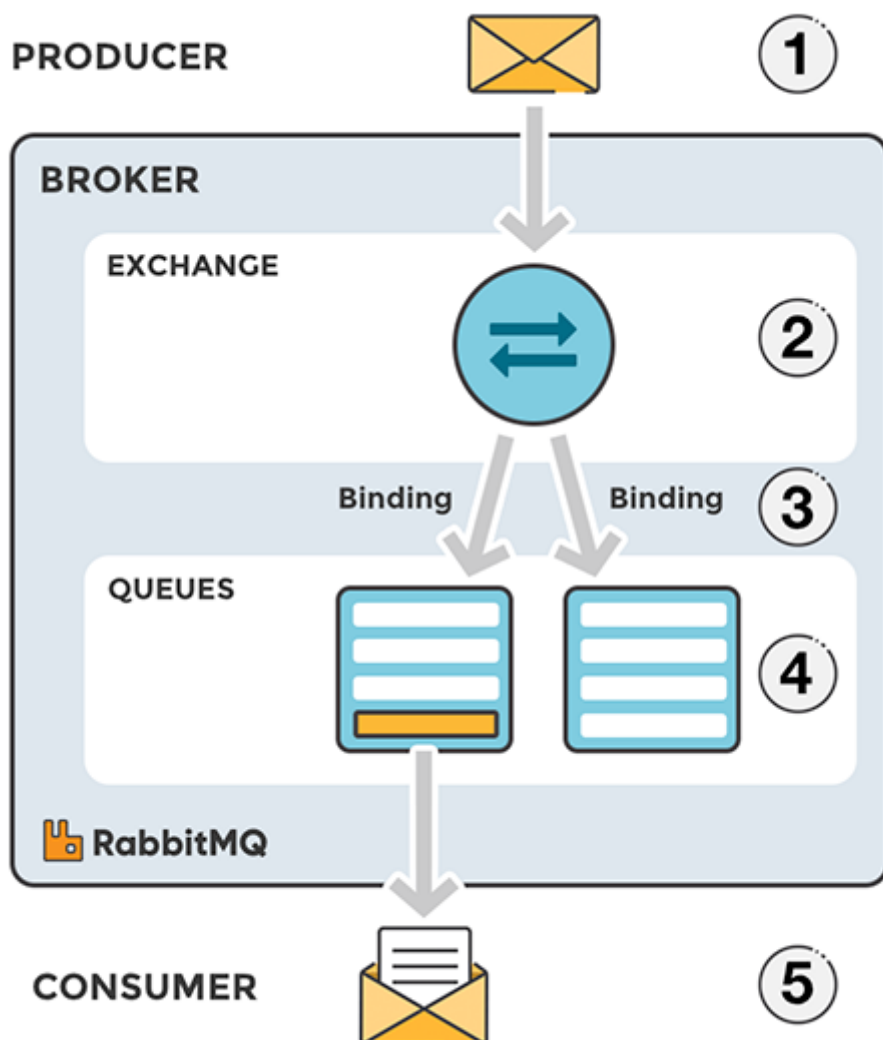
而如果我们引入RabbitMQ消息队列,订单数据可以发送到消息队列服务器,那么调用链路也就可以到此结束,订单系统则可以立即得到响应,整条链路的响应时间只有200毫秒左右

寻找外卖小哥的应用可以以异步的方式从消息队列接收订单消息,再执行耗时的寻找操作



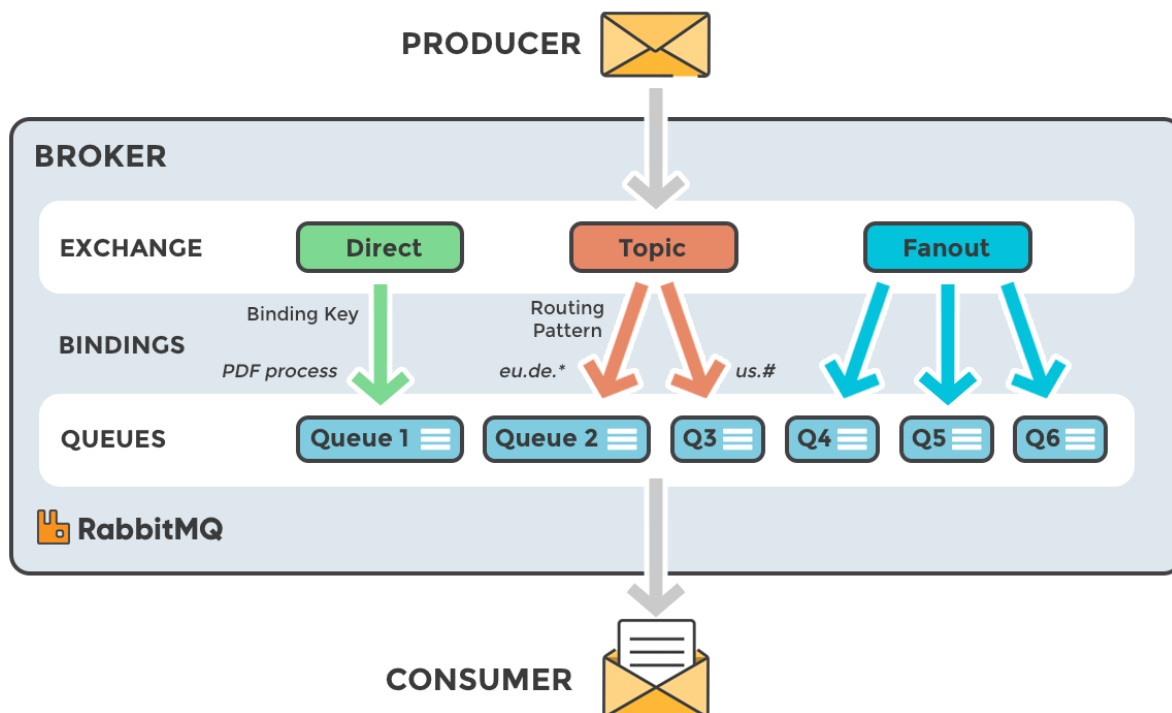
rabbitmq 基本概念

RabbitMQ是一种消息中间件，用于处理来自客户端的异步消息。服务端将要发送的消息放入到队列池中。接收端可以根据RabbitMQ配置的转发机制接收服务端发来的消息。RabbitMQ依据指定的转发规则进行消息的转发、缓冲和持久化操作，主要用在多服务器间或单服务器的子系统间进行通信，是分布式系统标准的配置。



Exchange

接受生产者发送的消息，并根据Binding规则将消息路由给服务器中的队列。ExchangeType决定了Exchange路由消息的行为。在RabbitMQ中，ExchangeType常用的有direct、Fanout和Topic三种。



Message Queue

消息队列。我们发送给RabbitMQ的消息最后都会到达各种queue，并且存储在其中(如果路由找不到相应的queue则数据会丢失)，等待消费者来取。

Binding Key

它表示的是Exchange与Message Queue是通过binding key进行联系的，这个关系是固定。

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。这个routing key需要与Exchange Type及binding key联合使用才能生效，我们的生产者只需要通过指定routing key来决定消息流向哪里。

rabbitmq安装

在centos7上安装rabbitmq

安装erlang语言库

RabbitMQ使用了Erlang开发语言，Erlang是为电话交换机开发的语言，天生自带高并发光环，和高可用特性

rabbitmq官方精简的Erlang语言包

- 0依赖rpm安装包

<https://github.com/rabbitmq/erlang-rpm>

Supported CentOS Versions

Please note the implicit OpenSSL/libcrypto dependency section above.

- CentOS 7
- CentOS 6

Release Artifacts

Yum repositories are available from [Package Cloud](#) and [Bintray](#) and (see repository setup instructions below).

RPM packages can be downloaded [from GitHub](#).

下载和安装

```
# 下载Erlang语言包
wget https://github.com/rabbitmq/erlang-rpm/releases/download/v21.2.6/erlang-21.2.6-1.el7.x86_64.rpm

# 安装Erlang
rpm -ivh erlang-21.2.6-1.el7.x86_64.rpm --force --nodeps
```

安装socat依赖

socat依赖包

- <https://pkgs.org/download/socat>

Socat Download for Linux (deb, rpm, tgz, txz)

Download socat linux packages for ALTLinux, Arch Linux, CentOS, Debian, Fedora, Ubuntu.

ALT Linux Sisyphus

Arch Linux

CentOS 7

CERT Forensics Tools x86_64

[socat-1.7.3.2-1.1.el7.x86_64.rpm](#)

CentOS x86_64

[socat-1.7.3.2-2.el7.x86_64.rpm](#) 

Lux all

[socat-1.7.3.2-5.el7.lux.x86_64.rpm](#)

Repoforge (RPMforge) x86_64

[socat-1.7.2.4-1.el7.rf.x86_64.rpm](#)

CentOS 6

- https://centos.pkgs.org/7/centos-x86_64/socat-1.7.3.2-2.el7.x86_64.rpm.html

Show Packages

Download

Type	URL
Mirror	mirror.centos.org
Binary Package	socat-1.7.3.2-2.el7.x86_64.rpm 
Source Package	socat-1.7.3.2-2.el7.src.rpm

Install Howto

Install socat rpm package:

```
# yum install socat
```

下载和安装

```
# 下载 socat rpm
wget http://mirror.centos.org/centos/7/os/x86_64/Packages/socat-1.7.3.2-2.el7.x86_64.rpm
```



```
# 安装 socat 依赖包
rpm -ivh socat-1.7.3.2-2.el7.x86_64.rpm
```

安装rabbitmq

rabbitmq安装包

- <http://www.rabbitmq.com/install-rpm.html#downloads>

Download the Server

In some cases it may easier to download the package and install it manually. The package can be downloaded from [GitHub](#).

Description	Download	
RPM for RHEL Linux 7.x, CentOS 7.x, Fedora 19+ (supports systemd)	rabbitmq-server-3.7.13-1.el7.noarch.rpm	(Signature)

下载和安装

```
# 下载 rpm 包
wget https://github.com/rabbitmq/rabbitmq-server/releases/download/v3.7.13/rabbitmq-server-3.7.13-1.el7.noarch.rpm

# 安装 rpm 包
rpm -ivh rabbitmq-server-3.7.13-1.el7.noarch.rpm
```

rabbitmq启动和停止命令

```
# 设置服务, 开机自动启动
chkconfig rabbitmq-server on

# 启动服务
service rabbitmq-server start

# 停止服务
service rabbitmq-server stop
```

rabbitmq管理界面

启用管理界面

```
# 开启管理界面插件
```

```
rabbitmq-plugins enable rabbitmq_management
```

```
# 防火墙打开 15672 管理端口
```

```
firewall-cmd --zone=public --add-port=15672/tcp --permanent
```

```
firewall-cmd --reload
```

访问

访问服务器的 15672 端口,例如:

<http://192.168.64.140:15672>

添加用户

添加用户

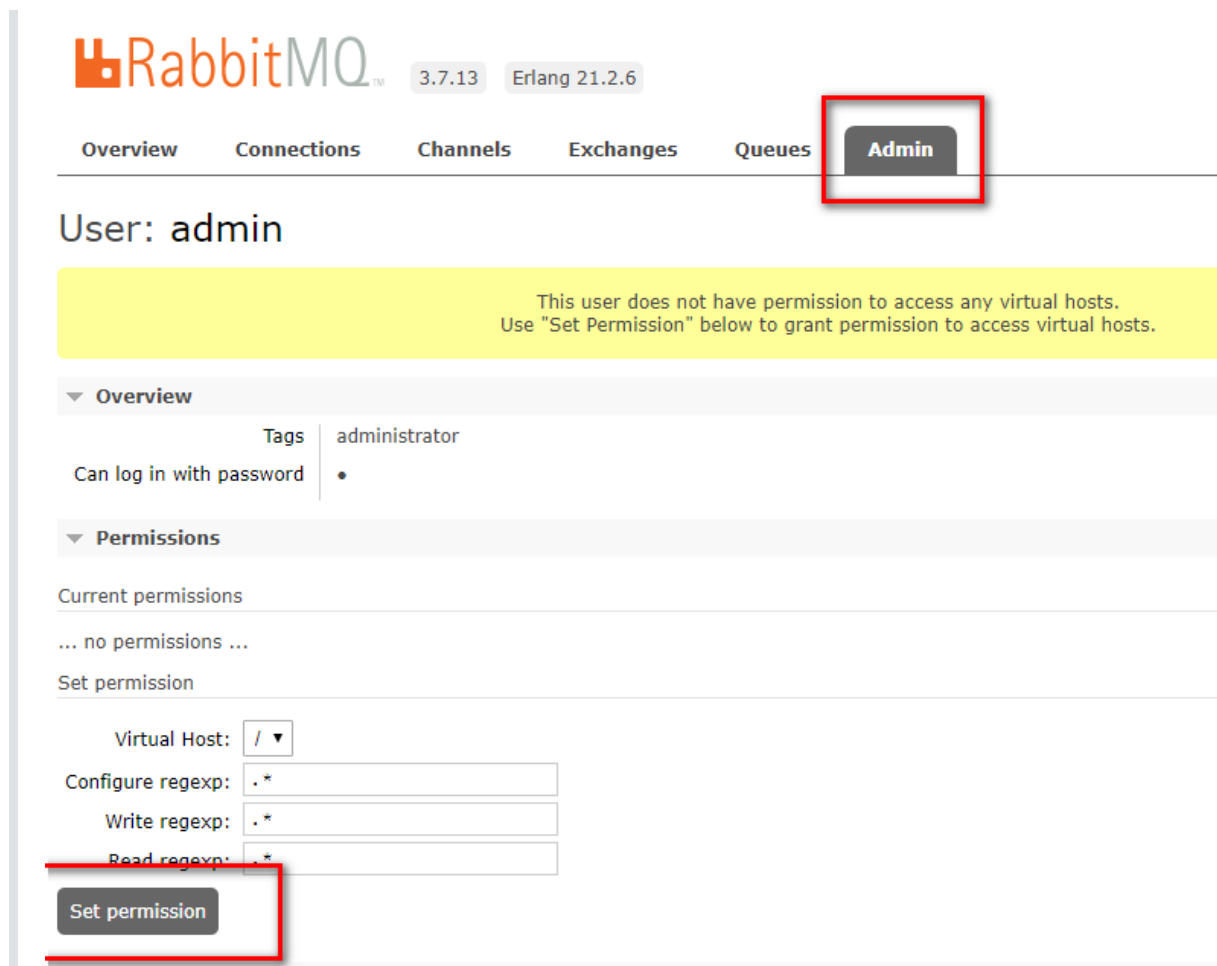
```
# 添加用户
```

```
rabbitmqctl add_user admin admin
```

```
# 新用户设置用户为超级管理员
```

```
rabbitmqctl set_user_tags admin administrator
```

设置访问权限



- 用户管理参考
<https://www.cnblogs.com/AloneSword/p/4200051.html>

开放客户端连接端口

```
# 打开客户端连接端口
firewall-cmd --zone=public --add-port=5672/tcp --permanent
firewall-cmd --reload
```

- 主要端口介绍
 - 4369 -- erlang发现口
 - 5672 -- client端通信口
 - 15672 -- 管理界面ui端口
 - 25672 -- server间内部通信口

rabbitmq六种工作模式

简单模式

RabbitMQ是一个消息中间件，你可以想象它是一个邮局。当你把信件放到邮箱里时，能够确信邮递员会正确地递送你的信件。RabbitMq就是一个邮箱、一个邮局和一个邮递员。

- 发送消息的程序是生产者
- 队列就代表一个邮箱。虽然消息会流经RabbitMQ和你的应用程序，但消息只能被存储在队列里。队列存储空间只受服务器内存和磁盘限制，它本质上是一个大的消息缓冲区。多个生产者可以向同一个队列发送消息，多个消费者也可以从同一个队列接收消息。
- 消费者等待从队列接收消息



pom.xml

添加 slf4j 依赖, 和 rabbitmq amqp 依赖

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.tedu</groupId>
  <artifactId>rabbitmq</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <dependencies>
    <dependency>
      <groupId>com.rabbitmq</groupId>
      <artifactId>amqp-client</artifactId>
      <version>5.4.3</version>
    </dependency>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-api</artifactId>
      <version>1.8.0-alpha2</version>
    </dependency>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-log4j12</artifactId>
      <version>1.8.0-alpha2</version>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.0</version>
        <configuration>
          <source>1.8</source>
          <target>1.8</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

```
        </configuration>
    </plugin>
</plugins>
</build>
</project>
```

生产者发送消息

```
package rabbitmq.simple;

import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;

public class Test1 {
    public static void main(String[] args) throws Exception {
        // 创建连接工厂, 并设置连接信息
        ConnectionFactory f = new ConnectionFactory();
        f.setHost("192.168.64.140");
        f.setPort(5672); // 可选, 5672是默认端口
        f.setUsername("admin");
        f.setPassword("admin");

        /*
         * 与rabbitmq服务器建立连接,
         * rabbitmq服务器端使用的是nio, 会复用tcp连接,
         * 并开辟多个信道与客户端通信
         * 以减轻服务器端建立连接的开销
         */
        Connection c = f.newConnection();
        // 建立信道, 即一个链接内创建多个信道
        Channel ch = c.createChannel();

        /*
         * 声明队列, 会在rabbitmq中创建一个队列
         * 如果已经创建过该队列, 就不能再使用其他参数来创建
         *
         * 参数含义:
         * -queue: 队列名称
         * -durable: 队列持久化, true表示RabbitMQ重启后队列仍存在
         * -exclusive: 排他, true表示限制仅当前连接可用
         * -autoDelete: 当最后一个消费者断开后, 是否删除队列
         * -arguments: 其他参数
         */
        ch.queueDeclare("helloworld", false, false, false, null);

        /*
         * 发布消息
         * 这里把消息向默认交换机发送.
         * 默认交换机隐含与所有队列绑定, routing key即为队列名称
         *
         * 参数含义:
         * -exchange: 交换机名称, 空串表示默认交换机("AMQP default"), 不能用 null
         * -routingKey: 对于默认交换机, 路由键就是目标队列名称
         */
    }
}
```

```

        * -props: 其他参数, 例如头信息
        * -body: 消息内容byte[]数组
        */
ch.basicPublish("", "helloworld", null, "Hello world!".getBytes());

System.out.println("消息已发送");
c.close();
    }
}

```

消费者接收消息

```

package rabbitmq.simple;

import java.io.IOException;
import java.util.concurrent.TimeoutException;

import com.rabbitmq.client.CancelCallback;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.DeliverCallback;
import com.rabbitmq.client.Delivery;

public class Test2 {
    public static void main(String[] args) throws Exception {
        //连接工厂
        ConnectionFactory f = new ConnectionFactory();
        f.setHost("192.168.64.140");
        f.setUsername("admin");
        f.setPassword("admin");
        //建立连接
        Connection c = f.newConnection();
        //建立信道
        Channel ch = c.createChannel();
        //声明队列, 如果该队列已经创建过, 则不会重复创建
        ch.queueDeclare("helloworld", false, false, false, null);
        System.out.println("等待接收数据");

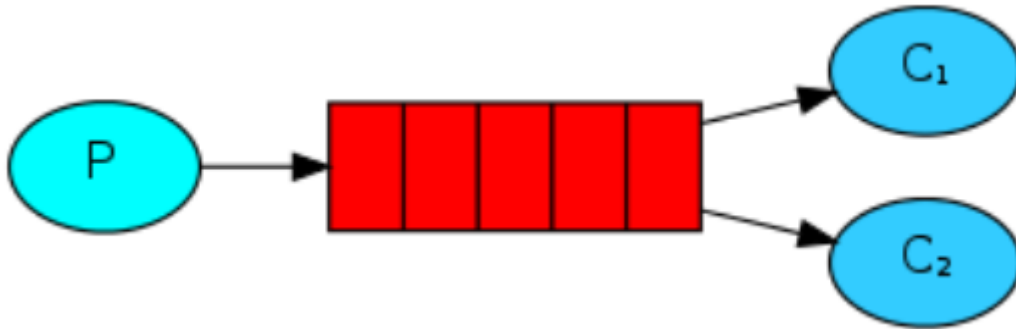
        //收到消息后用来处理消息的回调对象
        DeliverCallback callback = new DeliverCallback() {
            @Override
            public void handle(String consumerTag, Delivery message) throws IOException {
                String msg = new String(message.getBody(), "UTF-8");
                System.out.println("收到: "+msg);
            }
        };

        //消费者取消时的回调对象
        CancelCallback cancel = new CancelCallback() {
            @Override
            public void handle(String consumerTag) throws IOException {
            }
        };
    }
}

```

```
        ch.basicConsume("helloworld", true, callback, cancel);
    }
}
```

工作模式



工作队列(即任务队列)背后的主要思想是避免立即执行资源密集型任务，并且必须等待它完成。相反，我们将任务安排在稍后完成。

我们将任务封装为消息并将其发送到队列。后台运行的工作进程将获取任务并最终执行任务。当运行多个消费者时，任务将在它们之间分发。

使用任务队列的一个优点是能够轻松地并行工作。如果我们正在积压工作任务，我们可以添加更多工作进程，这样就可以轻松扩展。

生产者发送消息

这里模拟耗时任务,发送的消息中,每个点使工作进程暂停一秒钟,例如"Hello..."将花费3秒钟来处理

```
package rabbitmq.workqueue;

import java.util.Scanner;

import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;

public class Test1 {
    public static void main(String[] args) throws Exception {
        ConnectionFactory f = new ConnectionFactory();
        f.setHost("192.168.64.140");
        f.setPort(5672);
        f.setUsername("admin");
        f.setPassword("admin");

        Connection c = f.newConnection();
        Channel ch = c.createChannel();
        // 参数: queue, durable, exclusive, autoDelete, arguments
        ch.queueDeclare("helloworld", false, false, false, null);
    }
}
```

```

while (true) {
    //控制台输入的消息发送到rabbitmq
    System.out.print("输入消息: ");
    String msg = new Scanner(System.in).nextLine();
    //如果输入的是"exit"则结束生产者进程
    if ("exit".equals(msg)) {
        break;
    }
    //参数:exchange, routingKey, props, body
    ch.basicPublish("", "helloworld", null, msg.getBytes());
    System.out.println("消息已发送: "+msg);
}

c.close();
}
}

```

消费者接收消息

```

package rabbitmq.workqueue;

import java.io.IOException;
import java.util.concurrent.TimeoutException;

import com.rabbitmq.client.CancelCallback;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.DeliverCallback;
import com.rabbitmq.client.Delivery;

public class Test2 {
    public static void main(String[] args) throws Exception {
        ConnectionFactory f = new ConnectionFactory();
        f.setHost("192.168.64.140");
        f.setUsername("admin");
        f.setPassword("admin");
        Connection c = f.newConnection();
        Channel ch = c.createChannel();
        ch.queueDeclare("helloworld", false, false, false, null);
        System.out.println("等待接收数据");

        //收到消息后用来处理消息的回调对象
        DeliverCallback callback = new DeliverCallback() {
            @Override
            public void handle(String consumerTag, Delivery message) throws IOException {
                String msg = new String(message.getBody(), "UTF-8");
                System.out.println("收到: "+msg);

                //遍历字符串中的字符, 每个点使进程暂停一秒
                for (int i = 0; i < msg.length(); i++) {
                    if (msg.charAt(i) == '.') {
                        try {

```



```

        Thread.sleep(1000);
    } catch (InterruptedException e) {
    }
}
}
System.out.println("处理结束");
}
};

//消费者取消时的回调对象
CancelCallback cancel = new CancelCallback() {
    @Override
    public void handle(String consumerTag) throws IOException {
    }
};

ch.basicConsume("helloworld", true, callback, cancel);
}
}

```

运行测试

运行:

- 一个生产者
- 两个消费者

生产者发送多条消息, 如: 1,2,3,4,5. 两个消费者分别收到:

- 消费者一: 1,3,5
- 消费者二: 2,4

rabbitmq在所有消费者中轮询分发消息,把消息均匀地发送给所有消费者

消息确认

一个消费者接收消息后,在消息没有完全处理完时就挂掉了,那么这时会发生什么呢?

就现在的代码来说,rabbitmq把消息发送给消费者后,会立即删除消息,那么消费者挂掉后,它没来得及处理的消息就会丢失

如果生产者发送以下消息:

1.....
2
3
4
5

两个消费者分别收到:

- 消费者一: 1....., 3, 5

- 消费者二: 2, 4

当消费者一收到所有消息后,要花费7秒时间来处理第一条消息,这期间如果关闭该消费者,那么1未处理完成,3,5则没有被处理

我们并不想丢失任何消息, 如果一个消费者挂掉,我们想把它的任务消息派发给其他消费者

为了确保消息不会丢失, rabbitmq支持消息确认(回执)。当一个消息被消费者接收到并且执行完成后, 消费者会发送一个ack (acknowledgment) 给rabbitmq服务器, 告诉他我已经执行完成了, 你可以把这条消息删除了。

如果一个消费者没有返回消息确认就挂掉了(信道关闭, 连接关闭或者TCP链接丢失), rabbitmq就会明白, 这个消息没有被处理完成, rabbitmq就会把这条消息重新放入队列, 如果在这时有其他的消费者在线, 那么rabbitmq就会迅速的把这条消息传递给其他的消费者, 这样就确保了没有消息会丢失。

这里不存在消息超时, rabbitmq只在消费者挂掉时重新分派消息, 即使消费者花非常久的时间来处理消息也可以

手动消息确认默认是开启的, 前面的例子我们通过autoAck=true把它关闭了。我们现在要把它设置为false, 然后工作进程处理完意向任务时,发送一个消息确认(回执)。

```
package rabbitmq.workqueue;

import java.io.IOException;
import java.util.concurrent.TimeoutException;

import com.rabbitmq.client.CancelCallback;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.DeliverCallback;
import com.rabbitmq.client.Delivery;

public class Test2 {
    public static void main(String[] args) throws Exception {
        //连接工厂
        ConnectionFactory f = new ConnectionFactory();
        f.setHost("192.168.64.140");
        f.setUsername("admin");
        f.setPassword("admin");
        //建立连接
        Connection c = f.newConnection();
        //建立信道
        Channel ch = c.createChannel();
        //声明队列
        ch.queueDeclare("helloworld", false, false, false, null);
        System.out.println("等待接收数据");

        //收到消息后用来处理消息的回调对象
        DeliverCallback callback = new DeliverCallback() {
            @Override
            public void handle(String consumerTag, Delivery message) throws IOException {
```

```

        String msg = new String(message.getBody(), "UTF-8");
        System.out.println("收到: "+msg);
        for (int i = 0; i < msg.length(); i++) {
            if (msg.charAt(i)=='.') {
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                }
            }
        }
        System.out.println("处理结束");
        //发送回执
        ch.basicAck(message.getEnvelope().getDeliveryTag(), false);
    }
};

//消费者取消时的回调对象
CancelCallback cancel = new CancelCallback() {
    @Override
    public void handle(String consumerTag) throws IOException {
    }
};

//autoAck设置为false,则需要手动确认发送回执
ch.basicConsume("helloworld", false, callback, cancel);
}
}

```

使用以上代码，就算杀掉一个正在处理消息的工作进程也不会丢失任何消息，工作进程挂掉之后，没有确认的消息就会被自动重新传递。

忘记确认(ack)是一个常见的错误, 这样后果是很严重的, 由于未确认的消息不会被释放, rabbitmq会吃掉越来越多的内存

可以使用下面命令打印工作队列中未确认消息的数量

```
rabbitmqctl list_queues name messages_ready messages_unacknowledged
```

消息持久化

当rabbitmq关闭时, 我们队列中的消息仍然会丢失, 除非明确要求它不要丢失数据

要求rabbitmq不丢失数据要做如下两点: 把队列和消息都设置为可持久化(durable)

队列设置为可持久化, 可以在定义队列时指定参数durable为true

```

//第二个参数是持久化参数durable
ch.queueDeclare("helloworld", true, false, false, null);

```

由于之前我们已经定义过队列"hello"是不可持久化的, 对已存在的队列, rabbitmq不允许对其定义不同的参数, 否则会出错, 所以这里我们定义一个不同名字的队列"task_queue"

```
//定义一个新的队列, 名为 task_queue
//第二个参数是持久化参数 durable
ch.queueDeclare("task_queue", true, false, false, null);
```

生产者和消费者代码都要修改

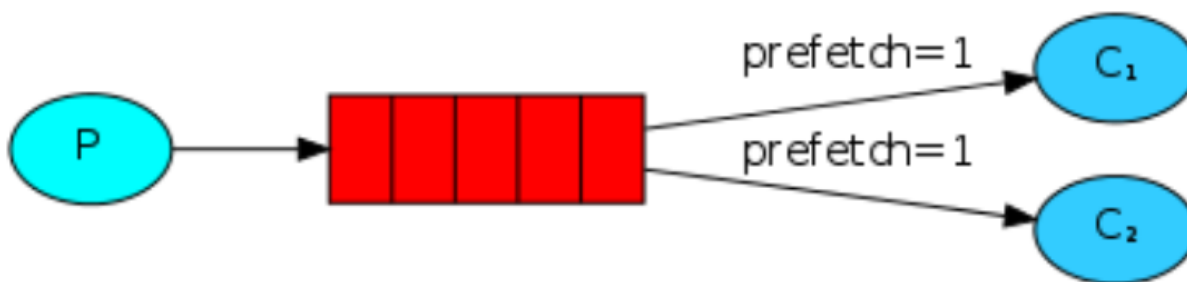
这样即使rabbitmq重新启动, 队列也不会丢失. 现在我们在设置队列中消息的持久化, 使用 `MessageProperties.PERSISTENT_TEXT_PLAIN` 参数

```
//第三个参数设置消息持久化
ch.basicPublish("", "task_queue",
    MessageProperties.PERSISTENT_TEXT_PLAIN,
    msg.getBytes());
```

合理地分发

rabbitmq会一次把多个消息分发给消费者, 这样可能造成有的消费者非常繁忙, 而其它消费者空闲. 而rabbitmq对此一无所知, 仍然会均匀的分发消息

我们可以使用 `setQos(1)` 方法, 这告诉rabbitmq一次只向消费者发送一条消息, 在返回确认回执前, 不要向消费者发送新消息. 而是把消息发给下一个空闲的消费者



下面是"工作模式"最终完成的生产者和消费者代码

生产者代码

```
package rabbitmq.workqueue;

import java.util.Scanner;

import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.MessageProperties;

public class Test3 {
```

```

public static void main(String[] args) throws Exception {
    ConnectionFactory f = new ConnectionFactory();
    f.setHost("192.168.64.140");
    f.setPort(5672);
    f.setUsername("admin");
    f.setPassword("admin");

    Connection c = f.newConnection();
    Channel ch = c.createChannel();

    // 第二个参数设置队列持久化
    ch.queueDeclare("task_queue", true, false, false, null);

    while (true) {
        System.out.print("输入消息: ");
        String msg = new Scanner(System.in).nextLine();
        if ("exit".equals(msg)) {
            break;
        }

        // 第三个参数设置消息持久化
        ch.basicPublish("", "task_queue", MessageProperties.PERSISTENT_TEXT_PLAIN,
msg.getBytes("UTF-8"));
        System.out.println("消息已发送: "+msg);
    }

    c.close();
}
}

```

消费者代码

```

package rabbitmq.workqueue;

import java.io.IOException;
import java.util.concurrent.TimeoutException;

import com.rabbitmq.client.CancelCallback;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.DeliverCallback;
import com.rabbitmq.client.Delivery;

public class Test4 {
    public static void main(String[] args) throws Exception {
        ConnectionFactory f = new ConnectionFactory();
        f.setHost("192.168.64.140");
        f.setUsername("admin");
        f.setPassword("admin");
        Connection c = f.newConnection();
        Channel ch = c.createChannel();

        // 第二个参数设置队列持久化
        ch.queueDeclare("task_queue", true, false, false, null);
    }
}

```

```

System.out.println("等待接收数据");

ch.basicQos(1); //一次只接收一条消息

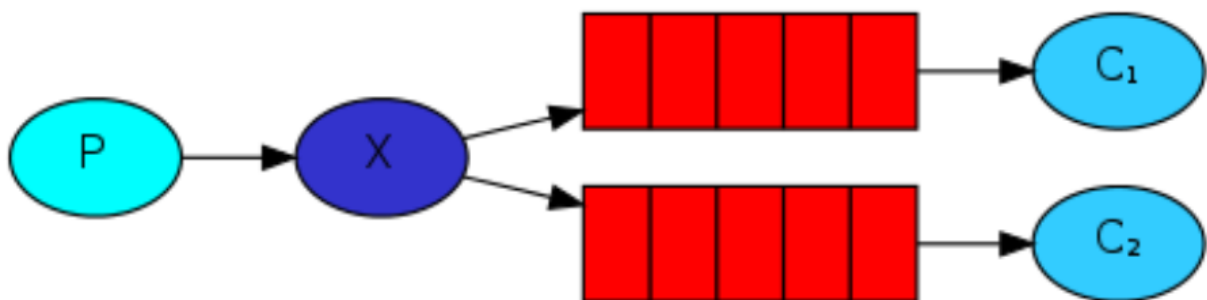
//收到消息后用来处理消息的回调对象
DeliverCallback callback = new DeliverCallback() {
    @Override
    public void handle(String consumerTag, Delivery message) throws IOException {
        String msg = new String(message.getBody(), "UTF-8");
        System.out.println("收到: "+msg);
        for (int i = 0; i < msg.length(); i++) {
            if (msg.charAt(i)=='.') {
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                }
            }
        }
        System.out.println("处理结束");
        //发送回执
        ch.basicAck(message.getEnvelope().getDeliveryTag(), false);
    }
};

//消费者取消时的回调对象
CancelCallback cancel = new CancelCallback() {
    @Override
    public void handle(String consumerTag) throws IOException {
    }
};

//autoAck设置为false,则需要手动确认发送回执
ch.basicConsume("task_queue", false, callback, cancel);
}
}

```

发布订阅模式



在前面的例子中，我们任务消息只交付给一个工作进程。在这部分，我们将做一些完全不同的事情——我们将向多个消费者传递同一条消息。这种模式称为“发布/订阅”。

为了说明该模式，我们将构建一个简单的日志系统。它将由两个程序组成——第一个程序将发出日志消息，第二个程序接收它们。

在我们的日志系统中，接收程序的每个运行副本都将获得消息。这样，我们就可以运行一个消费者并将日志保存到磁盘；同时我们可以运行另一个消费者在屏幕上打印日志。

最终，消息会被广播到所有消息接受者

Exchanges 交换机

RabbitMQ消息传递模型的核心思想是，生产者永远不会将任何消息直接发送到队列。实际上，通常生产者甚至不知道消息是否会被传递到任何队列。

相反，生产者只能向交换机(Exchange)发送消息。交换机是一个非常简单的东西。一边接收来自生产者的消息，另一边将消息推送到队列。交换器必须确切地知道如何处理它接收到的消息。它应该被添加到一个特定的队列中吗？它应该添加到多个队列中吗？或者它应该被丢弃。这些规则由exchange的类型定义。

有几种可用的交换类型:direct、topic、header和fanout。我们将关注最后一个——fanout。让我们创建一个这种类型的交换机，并称之为 logs: `ch.exchangeDeclare("logs", "fanout");`

fanout交换机非常简单。它只是将接收到的所有消息广播给它所知道的所有队列。这正是我们的日志系统所需要的。

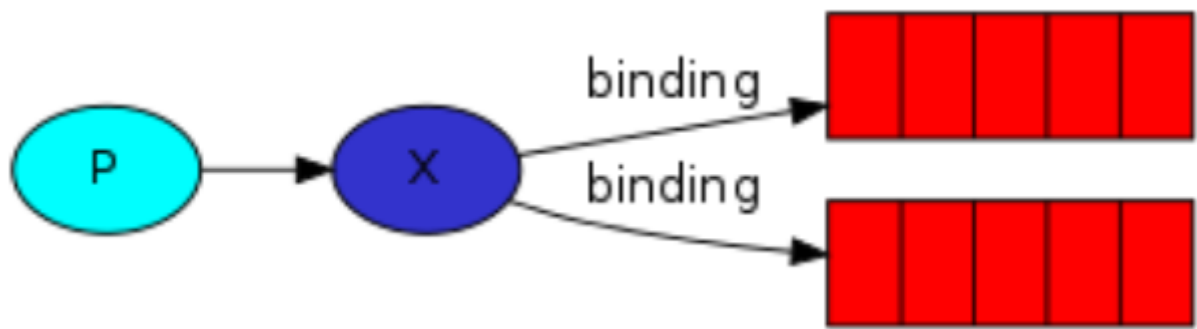
我们前面使用的队列具有特定的名称(还记得hello和task_queue吗?)能够为队列命名对我们来说至关重要——我们需要将工作进程指向同一个队列,在生产者和消费者之间共享队列。

但日志记录案例不是这种情况。我们想要接收所有的日志消息，而不仅仅是其中的一部分。我们还只对当前的最新消息感兴趣，而不是旧消息。

要解决这个问题，我们需要两件事。首先，每当我们连接到Rabbitmq时，我们需要一个新的空队列。为此，我们可以创建一个具有随机名称的队列，或者，更好的方法是让服务器为我们选择一个随机队列名称。其次，一旦断开与使用者的连接，队列就会自动删除。在Java客户端中，当我们不向queueDeclare()提供任何参数时，会创建一个具有生成名称的、非持久的、独占的、自动删除队列

```
// 自动生成队列名  
// 非持久, 独占, 自动删除  
String queueName = ch.queueDeclare().getQueue();
```

绑定 Bindings



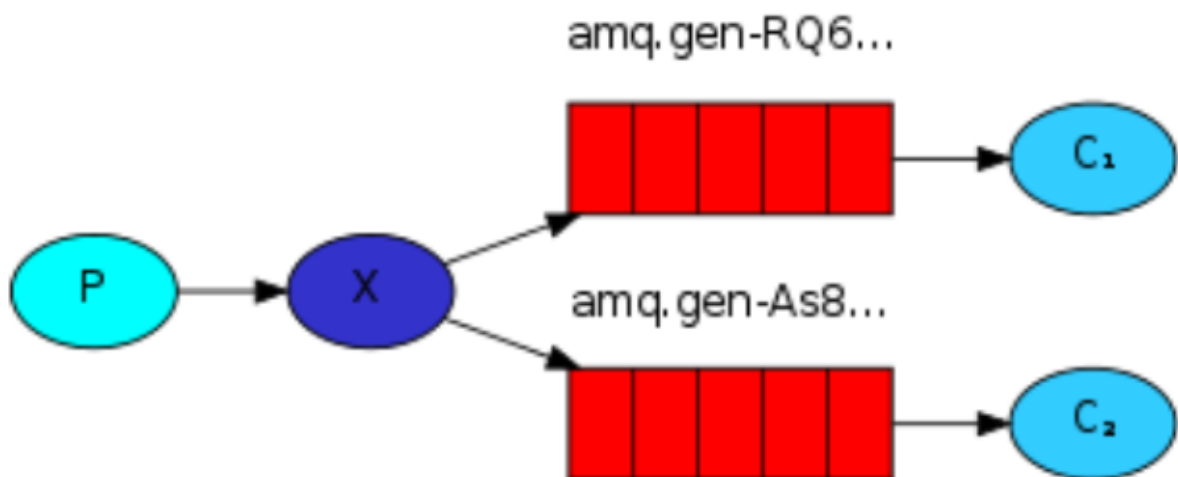
我们已经创建了一个fanout交换机和一个队列。现在我们需要告诉exchange向指定队列发送消息。exchange和队列之间的关系称为绑定。

```
// 指定的队列, 与指定的交换机关联起来
// 成为绑定 -- binding
// 第三个参数时 routingKey, 由于是fanout交换机, 这里忽略 routingKey
ch.queueBind(queueName, "logs", "");
```

现在, logs交换机将会向我们指定的队列添加消息

列出绑定关系:
rabbitmqctl list_bindings

完成的代码



生产者

生产者发出日志消息，看起来与前一教程没有太大不同。最重要的更改是，我们现在希望将消息发布到logs交换机，而不是无名的日志交换机。我们需要在发送时提供一个routingKey，但是对于fanout交换机类型，该值会被忽略。

```
package rabbitmq.publishsubscribe;

import java.util.Scanner;
```



```

import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;

public class Test3 {
    public static void main(String[] args) throws Exception {
        ConnectionFactory f = new ConnectionFactory();
        f.setHost("192.168.64.140");
        f.setPort(5672);
        f.setUsername("admin");
        f.setPassword("admin");

        Connection c = f.newConnection();
        Channel ch = c.createChannel();

        //定义名字为logs的交换机, 交换机类型为fanout
        //这一步是必须的, 因为禁止发布到不存在的交换。
        ch.exchangeDeclare("logs", "fanout");

        while (true) {
            System.out.print("输入消息: ");
            String msg = new Scanner(System.in).nextLine();
            if ("exit".equals(msg)) {
                break;
            }

            //第一个参数, 向指定的交换机发送消息
            //第二个参数, 不指定队列, 由消费者向交换机绑定队列
            //如果还没有队列绑定到交换机, 消息就会丢失,
            //但这对我们来说没有问题; 即使没有消费者接收, 我们也可以安全地丢弃这些信息。
            ch.basicPublish("logs", "", null, msg.getBytes("UTF-8"));
            System.out.println("消息已发送: "+msg);
        }

        c.close();
    }
}

```

消费者

如果还没有队列绑定到交换机, 消息就会丢失, 但这对我们来说没有问题; 如果还没有消费者在听, 我们可以安全地丢弃这些信息。ReceiveLogs.java代码:

```

package rabbitmq.publishsubscribe;

import java.io.IOException;

import com.rabbitmq.client.CancelCallback;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.DeliverCallback;
import com.rabbitmq.client.Delivery;

```

```

public class Test2 {
    public static void main(String[] args) throws Exception {
        ConnectionFactory f = new ConnectionFactory();
        f.setHost("192.168.64.140");
        f.setUsername("admin");
        f.setPassword("admin");
        Connection c = f.newConnection();
        Channel ch = c.createChannel();

        //定义名字为 Logs 的交换机, 它的类型是 fanout
        ch.exchangeDeclare("logs", "fanout");

        //自动生成队列名,
        //非持久, 独占, 自动删除
        String queueName = ch.queueDeclare().getQueue();

        //把该队列, 绑定到 Logs 交换机
        //对于 fanout 类型的交换机, routingKey 会被忽略
        ch.queueBind(queueName, "logs", null);

        System.out.println("等待接收数据");

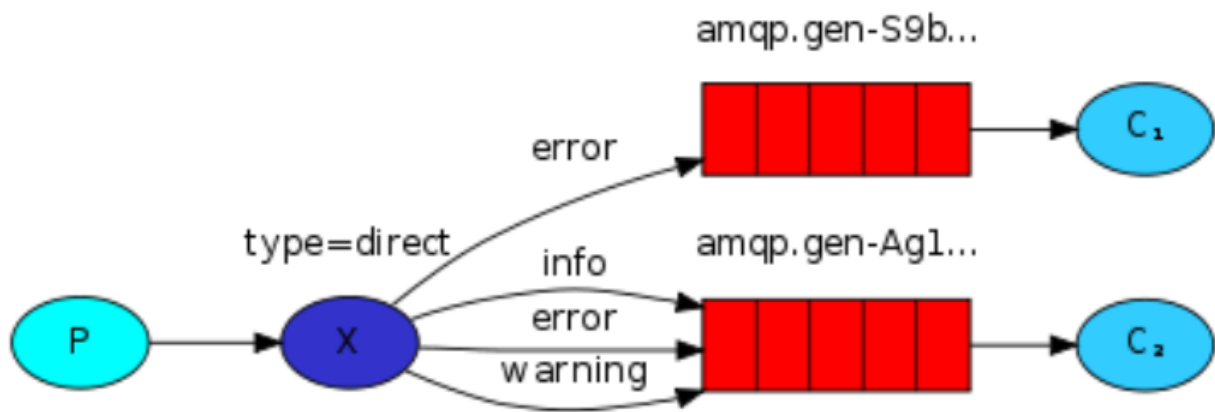
        //收到消息后用来处理消息的回调对象
        DeliverCallback callback = new DeliverCallback() {
            @Override
            public void handle(String consumerTag, Delivery message) throws IOException {
                String msg = new String(message.getBody(), "UTF-8");
                System.out.println("收到: "+msg);
            }
        };

        //消费者取消时的回调对象
        CancelCallback cancel = new CancelCallback() {
            @Override
            public void handle(String consumerTag) throws IOException {
            }
        };

        ch.basicConsume(queueName, true, callback, cancel);
    }
}

```

路由模式



在上一小节，我们构建了一个简单的日志系统。我们能够向多个接收者广播日志消息。

在这一节，我们将向其添加一个特性——我们将只订阅所有消息中的一部分。例如，我们只接收关键错误消息并保存到日志文件(以节省磁盘空间)，同时仍然能够在控制台上打印所有日志消息。

绑定 Bindings

在上一节，我们已经创建了队列与交换机的绑定。使用下面这样的代码：

```
ch.queueBind(queueName, "logs", "");
```

绑定是交换机和队列之间的关系。这可以简单地理解为：队列对来自此交换的消息感兴趣。

绑定可以使用额外的routingKey参数。为了避免与basic_publish参数混淆，我们将其称为bindingKey。这是我们如何创建一个键绑定：

```
ch.queueBind(queueName, EXCHANGE_NAME, "black");
```

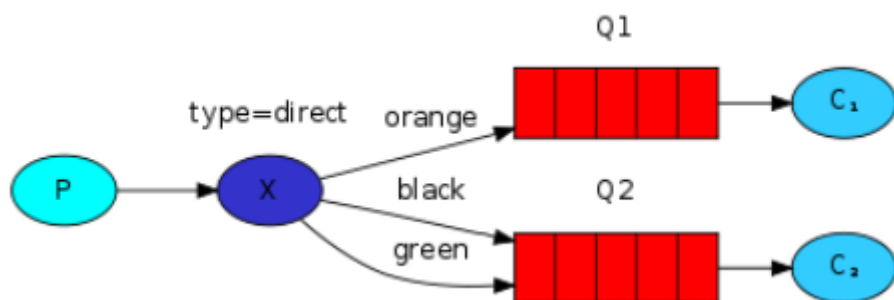
bindingKey的含义取决于交换机类型。我们前面使用的fanout交换机完全忽略它。

直连交换机 Direct exchange

上一节中的日志系统向所有消费者广播所有消息。我们希望扩展它，允许根据消息的严重性过滤消息。例如，我们希望将日志消息写入磁盘的程序只接收关键error，而不是在warning或info日志消息上浪费磁盘空间。

前面我们使用的是fanout交换机，这并没有给我们太多的灵活性——它只能进行简单的广播。

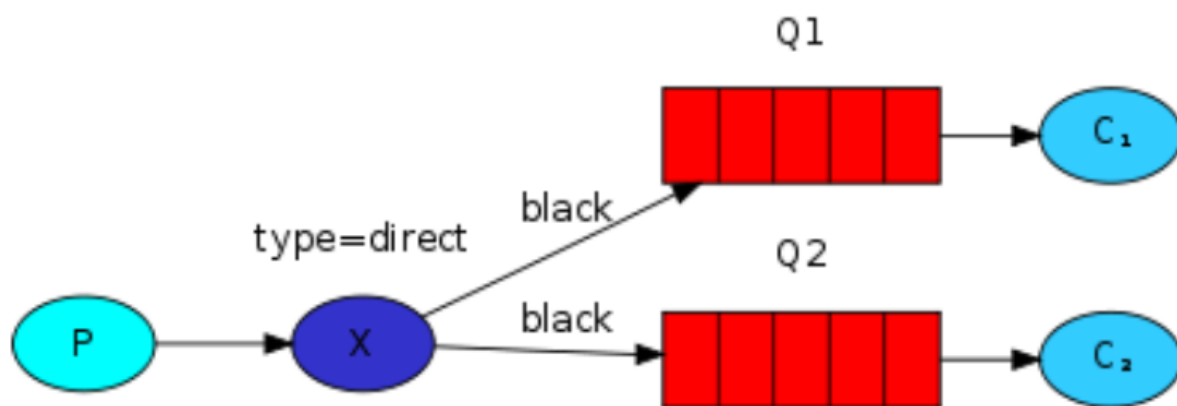
我们将用直连交换机(Direct exchange)代替。它背后的路由算法很简单——消息传递到bindingKey与routingKey完全匹配的队列。为了说明这一点，请考虑以下设置



其中我们可以看到直连交换机 `x`，它绑定了两个队列。第一个队列用绑定键 `orange` 绑定，第二个队列有两个绑定，一个绑定 `black`，另一个绑定键 `green`。

这样设置，使用路由键 `orange` 发布到交换器的消息将被路由到队列 `Q1`。带有 `black` 或 `green` 路由键的消息将转到 `Q2`。而所有其他消息都将被丢弃。

多重绑定 Multiple bindings



使用相同的bindingKey绑定多个队列是完全允许的。如图所示，可以使用binding key `black`将`X`与`Q1`和`Q2`绑定。在这种情况下，直连交换机的行为类似于fanout，并将消息广播给所有匹配的队列。一条路由键为`black`的消息将同时发送到`Q1`和`Q2`。

发送日志

我们将在日志系统中使用这个模型。我们把消息发送到一个Direct交换机，而不是fanout。我们将提供日志级别作为routingKey。这样，接收程序将能够选择它希望接收的级别。让我们首先来看发出日志。

和前面一样，我们首先需要创建一个exchange:

```
// 参数1: 交换机名
// 参数2: 交换机类型
ch.exchangeDeclare("direct_logs", "direct");
```

接着来看发送消息的代码

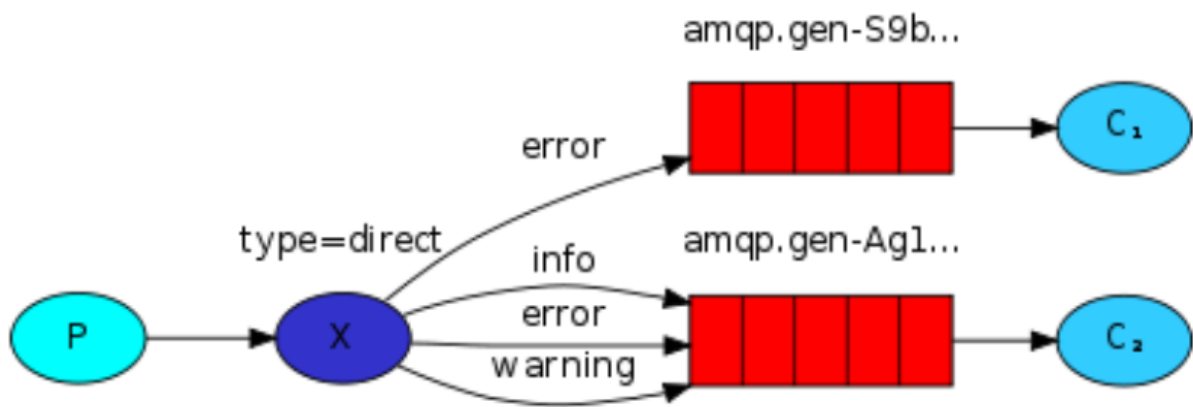
```
// 参数1: 交换机名
// 参数2: routingKey, 路由键, 这里我们用日志级别, 如"error", "info", "warning"
// 参数3: 其他配置属性
// 参数4: 发布的消息数据
ch.basicPublish("direct_logs", "error", null, message.getBytes());
```

订阅

接收消息的工作原理与前面章节一样, 但有一个例外——我们将为感兴趣的每个日志级别创建一个新的绑定, 示例代码如下:

```
ch.queueBind(queueName, "logs", "info");
ch.queueBind(queueName, "logs", "warning");
```

完整的代码



生产者

```
package rabbitmq.routing;

import java.util.Random;
import java.util.Scanner;

import com.rabbitmq.client.BuiltinExchangeType;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;

public class Test1 {
    public static void main(String[] args) throws Exception {
        String[] a = {"warning", "info", "error"};

        ConnectionFactory f = new ConnectionFactory();
        f.setHost("192.168.64.140");
        f.setPort(5672);
        f.setUsername("admin");
```

```

        f.setPassword("admin");

        Connection c = f.newConnection();
        Channel ch = c.createChannel();

        // 参数1: 交换机名
        // 参数2: 交换机类型
        ch.exchangeDeclare("direct_logs", BuiltinExchangeType.DIRECT);

        while (true) {
            System.out.print("输入消息: ");
            String msg = new Scanner(System.in).nextLine();
            if ("exit".equals(msg)) {
                break;
            }

            // 随机产生日志级别
            String level = a[new Random().nextInt(a.length)];

            // 参数1: 交换机名
            // 参数2: routingKey, 路由键, 这里我们用日志级别, 如"error", "info", "warning"
            // 参数3: 其他配置属性
            // 参数4: 发布的消息数据
            ch.basicPublish("direct_logs", level, null, msg.getBytes());
            System.out.println("消息已发送: "+level+" - "+msg);

        }

        c.close();
    }
}

```

消费者

```

package rabbitmq.routing;

import java.io.IOException;
import java.util.Scanner;

import com.rabbitmq.client.BuiltinExchangeType;
import com.rabbitmq.client.CancelCallback;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.DeliverCallback;
import com.rabbitmq.client.Delivery;

public class Test2 {
    public static void main(String[] args) throws Exception {
        ConnectionFactory f = new ConnectionFactory();
        f.setHost("192.168.64.140");
        f.setUsername("admin");
        f.setPassword("admin");
        Connection c = f.newConnection();
    }
}

```

```

Channel ch = c.createChannel();

//定义名字为 direct_logs 的交换机, 它的类型是 "direct"
ch.exchangeDeclare("direct_logs", BuiltinExchangeType.DIRECT);

//自动生成队列名,
//非持久, 独占, 自动删除
String queueName = ch.queueDeclare().getQueue();

System.out.println("输入接收的日志级别, 用空格隔开:");
String[] a = new Scanner(System.in).nextLine().split("\\s");

//把该队列, 绑定到 direct_logs 交换机
//允许使用多个 bindingKey
for (String level : a) {
    ch.queueBind(queueName, "direct_logs", level);
}

System.out.println("等待接收数据");

//收到消息后用来处理消息的回调对象
DeliverCallback callback = new DeliverCallback() {
    @Override
    public void handle(String consumerTag, Delivery message) throws IOException {
        String msg = new String(message.getBody(), "UTF-8");
        String routingKey = message.getEnvelope().getRoutingKey();
        System.out.println("收到: "+routingKey+" - "+msg);
    }
};

//消费者取消时的回调对象
CancelCallback cancel = new CancelCallback() {
    @Override
    public void handle(String consumerTag) throws IOException {
    }
};

ch.basicConsume(queueName, true, callback, cancel);
}
}

```

主题模式

在上一小节，我们改进了日志系统。我们没有使用只能进行广播的fanout交换机，而是使用Direct交换机，从而可以选择性接收日志。

虽然使用Direct交换机改进了我们的系统，但它仍然有局限性——它不能基于多个标准进行路由。

在我们的日志系统中，我们可能不仅希望根据级别订阅日志，还希望根据发出日志的源订阅日志。

这将给我们带来很大的灵活性——我们可能只想接收来自“cron”的关键错误，但也要接收来自“kern”的所有日志。

要在日志系统中实现这一点，我们需要了解更复杂的Topic交换机。

主题交换机 Topic exchange

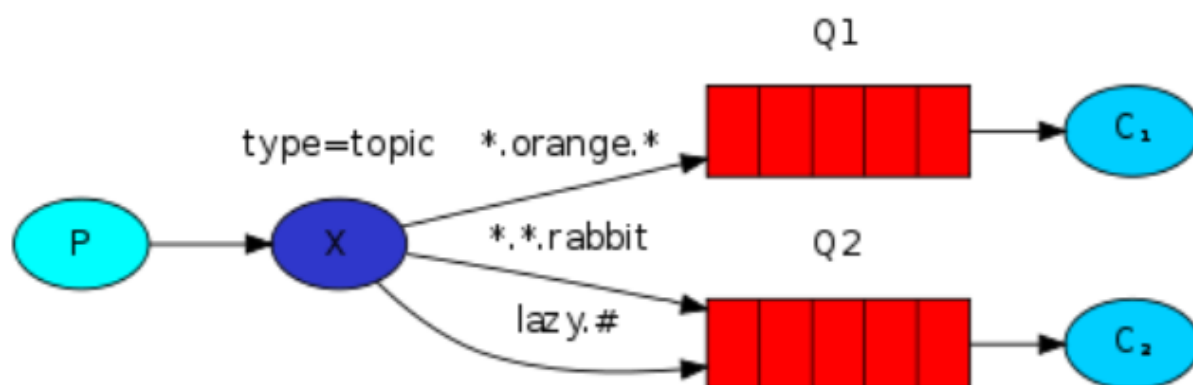
发送到Topic交换机的消息,它的routingKey,必须是由点分隔的多个单词。单词可以是任何东西，但通常是与消息相关的一些特性。几个有效的routingKey示

例: "stock.usd.nyse" 、 "nyse.vmw" 、 "quick.orange.rabbit" 。routingKey可以有任意多的单词，最多255个字节。

bindingKey也必须采用相同的形式。Topic交换机的逻辑与直连交换机类似——使用特定routingKey发送的消息将被传递到所有使用匹配bindingKey绑定的队列。bindingKey有两个重要的特殊点:

- * 可以通配单个单词。
- # 可以通配零个或多个单词。

用一个例子来解释这个问题是最简单的



在本例中，我们将发送描述动物的消息。这些消息将使用由三个单词(两个点)组成的routingKey发送。routingKey中的第一个单词表示速度，第二个是颜色,第三个是物种: "<速度>.<颜色>.<物种>"。

我们创建三个绑定:Q1与bindingKey "<速度>.<颜色>.*" 绑定。和Q2是 "<速度>.<颜色>.*.rabbit" 和 "lazy.#"。

这些绑定可概括为:

- Q1对所有橙色的动物感兴趣。
- Q2想接收关于兔子和慢速动物的所有消息。

将routingKey设置为" quick.orange.rabbit "的消息将被发送到两个队列。消息

" lazy.orange.elephant "也发送到它们两个。另外" quick.orange.fox "只会发到第一个队列，" lazy.brown.fox "只发给第二个。" lazy.pink.rabbit "将只被传递到第二个队列一次，即使它匹配两个绑定。" quick.brown.fox "不匹配任何绑定，因此将被丢弃。

如果我们违反约定，发送一个或四个单词的信息，比如" orange "或" quick.orange.male.rabbit "，会发生什么?这些消息将不匹配任何绑定，并将丢失。

另外, " lazy.orange.male.rabbit ", 即使它有四个单词, 也将匹配最后一个绑定, 并将被传递到第二个队列。

完成的代码

生产者

```
package rabbitmq.topic;

import java.util.Random;
import java.util.Scanner;

import com.rabbitmq.client.BuiltinExchangeType;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;

public class Test1 {
    public static void main(String[] args) throws Exception {
        ConnectionFactory f = new ConnectionFactory();
        f.setHost("192.168.64.140");
        f.setPort(5672);
        f.setUsername("admin");
        f.setPassword("admin");

        Connection c = f.newConnection();
        Channel ch = c.createChannel();

        // 参数1: 交换机名
        // 参数2: 交换机类型
        ch.exchangeDeclare("topic_logs", BuiltinExchangeType.TOPIC);

        while (true) {
            System.out.print("输入消息: ");
            String msg = new Scanner(System.in).nextLine();
            if ("exit".contentEquals(msg)) {
                break;
            }
            System.out.print("输入routingKey: ");
            String routingKey = new Scanner(System.in).nextLine();

            // 参数1: 交换机名
            // 参数2: routingKey, 路由键, 这里我们用日志级别, 如"error", "info", "warning"
            // 参数3: 其他配置属性
            // 参数4: 发布的消息数据
            ch.basicPublish("topic_logs", routingKey, null, msg.getBytes());

            System.out.println("消息已发送: "+routingKey+" - "+msg);
        }

        c.close();
    }
}
```

消费者

```
package rabbitmq.topic;

import java.io.IOException;
import java.util.Scanner;

import com.rabbitmq.client.BuiltinExchangeType;
import com.rabbitmq.client.CancelCallback;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.DeliverCallback;
import com.rabbitmq.client.Delivery;

public class Test2 {
    public static void main(String[] args) throws Exception {
        ConnectionFactory f = new ConnectionFactory();
        f.setHost("192.168.64.140");
        f.setUsername("admin");
        f.setPassword("admin");
        Connection c = f.newConnection();
        Channel ch = c.createChannel();

        ch.exchangeDeclare("topic_logs", BuiltinExchangeType.TOPIC);

        //自动生成队列名,
        //非持久,独占,自动删除
        String queueName = ch.queueDeclare().getQueue();

        System.out.println("输入bindingKey,用空格隔开:");
        String[] a = new Scanner(System.in).nextLine().split("\\s");

        //把该队列,绑定到 topic_logs 交换机
        //允许使用多个 bindingKey
        for (String bindingKey : a) {
            ch.queueBind(queueName, "topic_logs", bindingKey);
        }

        System.out.println("等待接收数据");

        //收到消息后用来处理消息的回调对象
        DeliverCallback callback = new DeliverCallback() {
            @Override
            public void handle(String consumerTag, Delivery message) throws IOException {
                String msg = new String(message.getBody(), "UTF-8");
                String routingKey = message.getEnvelope().getRoutingKey();
                System.out.println("收到: "+routingKey+" - "+msg);
            }
        };

        //消费者取消时的回调对象
        CancelCallback cancel = new CancelCallback() {
            @Override
            public void handle(String consumerTag) throws IOException {
            }
        };
    }
}
```

```
};

    ch.basicConsume(queueName, true, callback, cancel);
}
}
```

RPC模式

如果我们需要在远程电脑上运行一个方法，并且还要等待一个返回结果该怎么办？这和前面的例子不太一样，这种模式我们通常称为远程过程调用，即RPC。

在本节中，我们将会学习使用RabbitMQ去搭建一个RPC系统：一个客户端和一个可以升级(扩展)的RPC服务器。为了模拟一个耗时任务，我们将创建一个返回斐波那契数列的虚拟的RPC服务。

客户端

在客户端定义一个RPCClient类,并定义一个call()方法,这个方法发送一个RPC请求,并等待接收响应结果

```
RPCClient client = new RPCClient();
String result = client.call("4");
System.out.println( "第四个斐波那契数是: " + result);
```

回调队列 Callback Queue

使用RabbitMQ去实现RPC很容易。一个客户端发送请求信息，并得到一个服务器端回复的响应信息。为了得到响应信息，我们需要在请求的时候发送一个“回调”队列地址。我们可以使用默认队列。下面是示例代码：

```
// 定义回调队列,
// 自动生成队列名, 非持久, 独占, 自动删除
callbackQueueName = ch.queueDeclare().getQueue();

// 用来设置回调队列的参数对象
BasicProperties props = new BasicProperties
                        .Builder()
                        .replyTo(callbackQueueName)
                        .build();

// 发送调用消息
ch.basicPublish("", "rpc_queue", props, message.getBytes());
```

消息属性 Message Properties

AMQP 0-9-1协议定义了消息的14个属性。大部分属性很少使用，下面是比较常用的4个：

`deliveryMode` :将消息标记为持久化(值为2)或非持久化(任何其他值)。

`contentType` :用于描述mime类型。例如，对于经常使用的JSON格式，将此属性设置为: `application/json` 。

`replyTo` :通常用于指定回调队列。

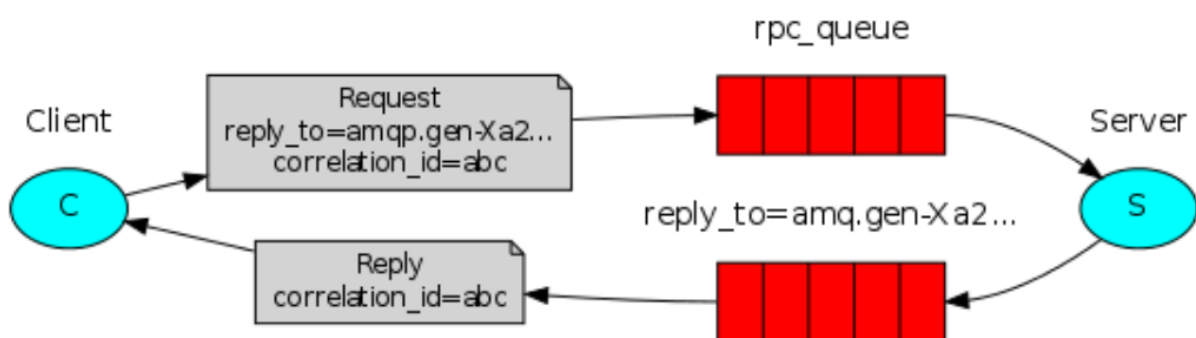
`correlationId` :将RPC响应与请求关联起来非常有用。

关联id (correlationId):

在上面的代码中，我们会为每个RPC请求创建一个回调队列。这是非常低效的，这里还有一个更好的方法:让我们为每个客户端创建一个回调队列。

这就提出了一个新的问题，在队列中得到一个响应时，我们不清楚这个响应所对应的是哪一条请求。这时候就需要使用关联id (correlationId) 。我们将为每一条请求设置唯一的id值。稍后，当我们在回调队列里收到一条消息的时候，我们将查看它的id属性，这样我们就可以匹配对应的请求和响应。如果我们发现了一个未知的id值，我们可以安全的丢弃这条消息，因为它不属于我们的请求。

小结



RPC的工作方式是这样的:

- 对于RPC请求，客户端发送一条带有两个属性的消息:replyTo,设置为仅为请求创建的匿名独占队列,和correlationId,设置为每个请求的惟一id值。
- 请求被发送到rpc_queue队列。
- RPC工作进程(即:服务器)在队列上等待请求。当一个请求出现时，它执行任务,并使用replyTo字段中的队列将结果发回客户机。
- 客户机在回应消息队列上等待数据。当消息出现时，它检查correlationId属性。如果匹配请求中的值，则向程序返回该响应数据。

完成的代码

服务器端

```
package rabbitmq.rpc;

import java.io.IOException;
import java.util.Random;
```

```

import java.util.Scanner;

import com.rabbitmq.client.AMQP;
import com.rabbitmq.client.BuiltinExchangeType;
import com.rabbitmq.client.CancelCallback;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.DeliverCallback;
import com.rabbitmq.client.Delivery;
import com.rabbitmq.client.AMQP.BasicProperties;

public class RPCServer {
    public static void main(String[] args) throws Exception {
        ConnectionFactory f = new ConnectionFactory();
        f.setHost("192.168.64.140");
        f.setPort(5672);
        f.setUsername("admin");
        f.setPassword("admin");

        Connection c = f.newConnection();
        Channel ch = c.createChannel();
        /*
         * 定义队列 rpc_queue, 将从它接收请求信息
         *
         * 参数:
         * 1. queue, 对列名
         * 2. durable, 持久化
         * 3. exclusive, 排他
         * 4. autoDelete, 自动删除
         * 5. arguments, 其他参数属性
         */
        ch.queueDeclare("rpc_queue", false, false, false, null);
        ch.queuePurge("rpc_queue");//清除队列中的内容

        ch.basicQos(1);//一次只接收一条消息

        //收到请求消息后的回调对象
        DeliverCallback deliverCallback = new DeliverCallback() {
            @Override
            public void handle(String consumerTag, Delivery message) throws IOException {
                //处理收到的数据(要求第几个斐波那契数)
                String msg = new String(message.getBody(), "UTF-8");
                int n = Integer.parseInt(msg);
                //求出第n个斐波那契数
                int r = fbnq(n);
                String response = String.valueOf(r);

                //设置发回响应的id, 与请求id一致, 这样客户端可以把该响应与它的请求进行对应
                BasicProperties replyProps = new BasicProperties.Builder()
                    .correlationId(message.getProperties().getCorrelationId())
                    .build();
                /*
                 * 发送响应消息
                 * 1. 默认交换机
                 * 2. 由客户端指定的, 用来传递响应消息的队列名
                */
            }
        };
    }
}

```

```

        * 3. 参数(关联id)
        * 4. 发回的响应消息
        */
        ch.basicPublish("",message.getProperties().getReplyTo(), replyProps,
response.getBytes("UTF-8"));
        //发送确认消息
        ch.basicAck(message.getEnvelope().getDeliveryTag(), false);
    }
};

//
CancelCallback cancelCallback = new CancelCallback() {
    @Override
    public void handle(String consumerTag) throws IOException {
    }
};

//消费者开始接收消息, 等待从 rpc_queue接收请求消息, 不自动确认
ch.basicConsume("rpc_queue", false, deliverCallback, cancelCallback);
}

protected static int fbnq(int n) {
    if(n == 1 || n == 2) return 1;

    return fbnq(n-1)+fbnq(n-2);
}
}

```

客户端

```

package rabbitmq.rpc;

import java.io.IOException;
import java.util.Scanner;
import java.util.UUID;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

import com.rabbitmq.client.BuiltinExchangeType;
import com.rabbitmq.client.CancelCallback;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.DeliverCallback;
import com.rabbitmq.client.Delivery;
import com.rabbitmq.client.AMQP.BasicProperties;

public class RPCClient {
    Connection con;
    Channel ch;

    public RPCClient() throws Exception {
        ConnectionFactory f = new ConnectionFactory();
        f.setHost("192.168.64.140");
    }
}

```

```

        f.setUsername("admin");
        f.setPassword("admin");
        con = f.newConnection();
        ch = con.createChannel();
    }

    public String call(String msg) throws Exception {
        //自动生成队列名, 非持久, 独占, 自动删除
        String replyQueueName = ch.queueDeclare().getQueue();
        //生成关联id
        String corrId = UUID.randomUUID().toString();

        //设置两个参数:
        //1. 请求和响应的关联id
        //2. 传递响应数据的queue
        BasicProperties props = new BasicProperties.Builder()
            .correlationId(corrId)
            .replyTo(replyQueueName)
            .build();
        //向 rpc_queue 队列发送请求数据, 请求第n个斐波那契数
        ch.basicPublish("", "rpc_queue", props, msg.getBytes("UTF-8"));

        //用来保存结果的阻塞集合, 取数据时, 没有数据会暂停等待
        BlockingQueue<String> response = new ArrayBlockingQueue<String>(1);

        //接收响应数据的回调对象
        DeliverCallback deliverCallback = new DeliverCallback() {
            @Override
            public void handle(String consumerTag, Delivery message) throws IOException {
                //如果响应消息的关联id, 与请求的关联id相同, 我们来处理这个响应数据
                if (message.getProperties().getCorrelationId().contentEquals(corrId)) {
                    //把收到的响应数据, 放入阻塞集合
                    response.offer(new String(message.getBody(), "UTF-8"));
                }
            }
        };

        CancelCallback cancelCallback = new CancelCallback() {
            @Override
            public void handle(String consumerTag) throws IOException {
            }
        };

        //开始从队列接收响应数据
        ch.basicConsume(replyQueueName, true, deliverCallback, cancelCallback);
        //返回保存在集合中的响应数据
        return response.take();
    }

    public static void main(String[] args) throws Exception {
        RPCCClient client = new RPCCClient();
        while (true) {
            System.out.print("求第几个斐波那契数:");
            int n = new Scanner(System.in).nextInt();
            String r = client.call(""+n);
            System.out.println(r);
        }
    }

```

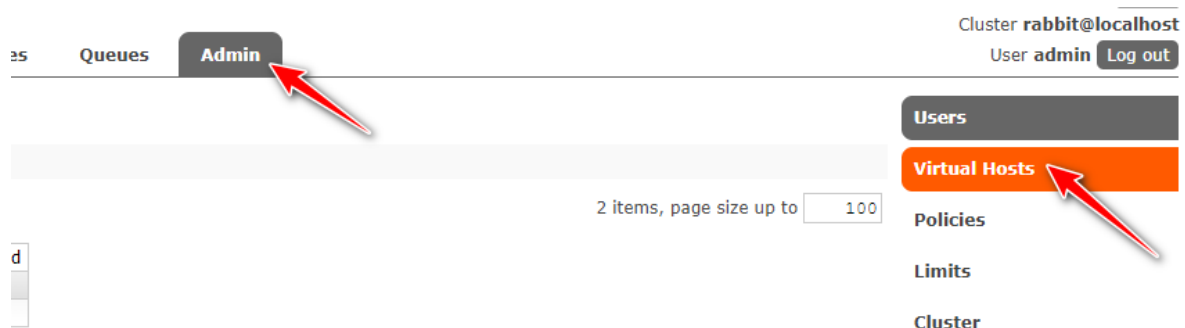
```
}  
}
```

virtual host

在RabbitMQ中叫做虚拟消息服务器VirtualHost，每个VirtualHost相当于一个相对独立的RabbitMQ服务器，每个VirtualHost之间是相互隔离的。exchange、queue、message不能互通

创建virtual host: /pd

- 进入虚拟机管理界面



- 添加新的虚拟机'/pd',名称必须以"/"开头

Virtual Hosts

▼ All virtual hosts

Filter: ☐ Regex ?

Overview			Messages			Network
Name	Users ?	State	Ready	Unacked	Total	From c
/	admin, guest	■ running	4	0	4	0B/s

▼ Add a new virtual host

Name: *

Add virtual host

- 查看添加的结果

Overview			Messages			Net
Name	Users ?	State	Ready	Unacked	Total	Fre
/	admin, guest	■ running	4	0	4	0B,
/pd	admin	■ running	NaN	NaN	NaN	

设置虚拟机的用户访问权限

点击 /pd 虚拟主机, 设置用户 admin 对它的访问权限

Virtual Host: /pd

► Overview

▼ Permissions

Current permissions

User	Configure regexp	Write regexp	Read regexp	
admin	.*	.*	.*	Clear

Set permission

User

admin ▼

Configure regexp: .*

Write regexp: .*

Read regexp: .*

Set permission

▼ Topic permissions

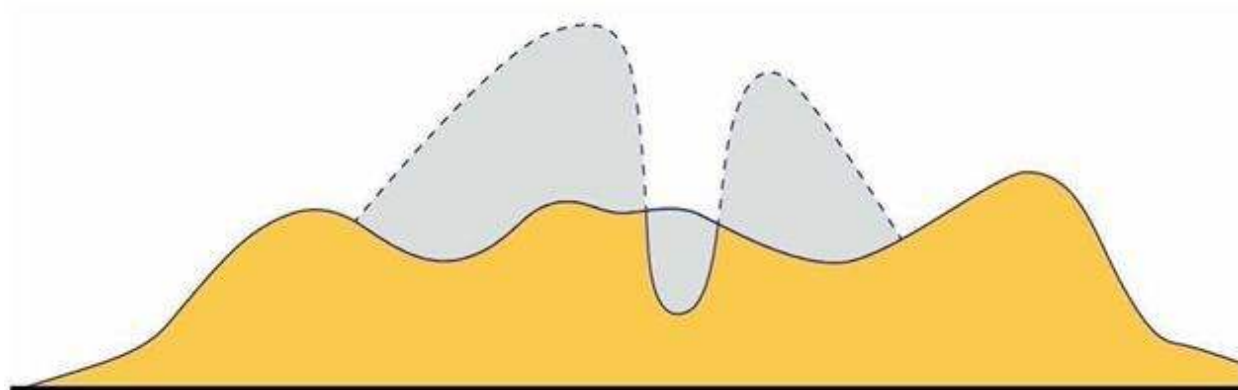
Current topic permissions

拼多多商城整合 rabbitmq

当用户下订单时,我们的业务系统直接与数据库通信,把订单保存到数据库中

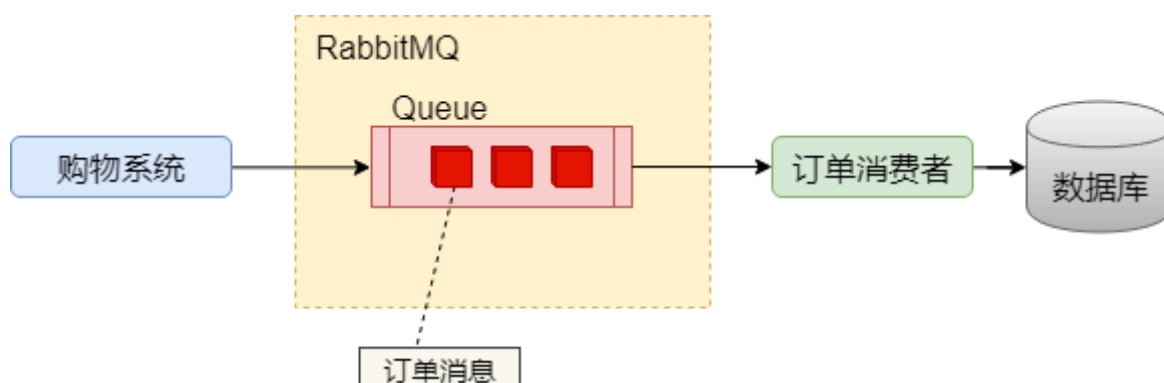
当系统流量突然激增,大量的订单压力,会拖慢业务系统和数据库系统

我们需要应对流量峰值,让流量曲线变得平缓,如下图



订单存储的解耦

为了进行流量削峰,我们引入 rabbitmq 消息队列,当购物系统产生订单后,可以把订单数据发送到消息队列;而订单消费者应用从消息队列接收订单消息,并把订单保存到数据库



这样,当流量激增时,大量订单会暂存在rabbitmq中,而订单消费者可以从容地从消息队列慢慢接收订单,向数据库保存

生产者-发送订单

pom.xml 添加依赖

spring提供了更方便的消息队列访问接口,它对RabbitMQ的客户端API进行了封装,使用起来更加方便

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

application.yml

添加RabbitMQ的连接信息

```
spring:
  rabbitmq:
    host: 192.168.64.140
    port: 5672
    virtualHost: /pd
    username: admin
    password: admin
```

修改主程序 RunPdAPP

在主程序中添加下面的方法创建Queue实例

当创建RabbitMQ连接和信道后, Spring的RabbitMQ工具会自动在服务器中创建队列, 代码在 `RabbitAdmin.declareQueues()` 方法中

```
@Bean
public Queue getQueue() {
    Queue q = new Queue("orderQueue", true);
    return q;
}
```

修改 OrderServiceImpl

```
//RabbitAutoConfiguration中创建了AmpqTemplate实例
@Autowired
AmpqTemplate amqpTemplate;

//saveOrder原来的数据库访问代码全部注释, 添加rabbitmq消息发送代码
public String saveOrder(PdOrder pdOrder) throws Exception {
    String orderId = generateId();
    pdOrder.setOrderId(orderId);

    amqpTemplate.convertAndSend("orderQueue", pdOrder);
    return orderId;

    //      String orderId = generateId();
    //      pdOrder.setOrderId(orderId);
    //
    //
    //      PdShipping pdShipping =
    pdShippingMapper.selectByPrimaryKey(pdOrder.getAddId());
    //      pdOrder.setShippingName(pdShipping.getReceiverName());
    //      pdOrder.setShippingCode(pdShipping.getReceiverAddress());
    //      pdOrder.setStatus(1);
    //      pdOrder.setPaymentType(1);
    //      pdOrder.setPostFee(100);
    //      pdOrder.setCreateTime(new Date());
    //
```

```



        //      double payment = 0;
        //      List<ItemVO> itemVOs =
selectCartItemByUserIdAndItemIds(pdOrder.getUserId(), pdOrder.getItemIdList());
        //      for (ItemVO itemVO : itemVOs) {
        //          PdOrderItem pdOrderItem = new PdOrderItem();
        //          String id = generateId();
        //          //String id="2";
        //          pdOrderItem.setId(id);
        //          pdOrderItem.setOrderId(orderId);
        //          pdOrderItem.setItemId("'" + itemVO.getPdItem().getId());
        //          pdOrderItem.setTitle(itemVO.getPdItem().getTitle());
        //          pdOrderItem.setPrice(itemVO.getPdItem().getPrice());
        //          pdOrderItem.setNum(itemVO.getPdCartItem().getNum());
        //
        //          payment = payment + itemVO.getPdCartItem().getNum() *
itemVO.getPdItem().getPrice();
        //          pdOrderItemMapper.insert(pdOrderItem);
        //      }
        //      pdOrder.setPayment(payment);
        //      pdOrderMapper.insert(pdOrder);
        //      return orderId;
    }

```

消费者-接收订单,并保存到数据库

pd-web项目复制为pd-order-consumer

```

>  > pd-order-consumer
>  > pd-web

```

修改 application.yml

把端口修改成 81

```

server:
  port: 81

spring:
  datasource:
    type: com.alibaba.druid.pool.DruidDataSource
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql://localhost:3306/pd_store?useUnicode=true&characterEncoding=utf8
    username: root
    password: root

  rabbitmq:
    host: 192.168.64.140
    port: 5672
    virtualHost: /pd
    username: admin

```

```
password: admin

mybatis:
  #typeAliasesPackage: cn.tedu.ssm.pojo
  mapperLocations: classpath:com.pd.mapper/*.xml

logging:
  level:
    cn.tedu.ssm.mapper: debug
```

删除无关代码

pd-order-consumer项目只需要从 RabbitMQ 接收订单数据, 再保存到数据库即可, 所以项目中只需要保留这部分代码

- 删除 com.pd.controller 包
- 删除 com.pd.payment.utils 包
- 删除无关的 Service,只保留 OrderService 和 OrderServiceImpl

新建 OrderConsumer

```
package com.pd;

import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import com.pd.pojo.PdOrder;
import com.pd.service.OrderService;

@Component
public class OrderConsumer {
    // 收到订单数据后, 会调用订单的业务代码, 把订单保存到数据库
    @Autowired
    private OrderService orderService;

    // 添加该注解后, 会从指定的orderQueue接收消息,
    // 并把数据转为 PdOrder 实例传递到此方法
    @RabbitListener(queues="orderQueue")
    public void save(PdOrder pdOrder)
    {
        System.out.println("消费者");
        System.out.println(pdOrder.toString());
        try {
            orderService.saveOrder(pdOrder);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

修改 OrderServiceImpl 的 saveOrder() 方法

```
public String saveOrder(PdOrder pdOrder) throws Exception {  
    //      String orderId = generateId();  
    //      pdOrder.setOrderId(orderId);  
    //  
    //      amqpTemplate.convertAndSend("orderQueue", pdOrder);  
    //      return orderId;  
    //  
    //  
    //      String orderId = generateId();  
    //      pdOrder.setOrderId(orderId);  
  
    //从RabbitMQ接收的订单数据,  
    //已经在上游订单业务中生成过id,这里不再重新生成id  
    //直接获取该订单的id  
    String orderId = pdOrder.getOrderId();  
  
    PdShipping pdShipping = pdShippingMapper.selectByPrimaryKey(pdOrder.getAddId());  
    pdOrder.setShippingName(pdShipping.getReceiverName());  
    pdOrder.setShippingCode(pdShipping.getReceiverAddress());  
    pdOrder.setStatus(1);//  
    pdOrder.setPaymentType(1);  
    pdOrder.setPostFee(100);  
    pdOrder.setCreateTime(new Date());  
  
    double payment = 0;  
    List<ItemVO> itemVOs = selectCartItemByUseridAndItemIds(pdOrder.getUserId(),  
pdOrder.getItemIdList());  
    for (ItemVO itemVO : itemVOs) {  
        PdOrderItem pdOrderItem = new PdOrderItem();  
        String id = generateId();  
        //String id="2";  
        pdOrderItem.setId(id);  
        pdOrderItem.setOrderId(orderId);  
        pdOrderItem.setItemId("" + itemVO.getPdItem().getId());  
        pdOrderItem.setTitle(itemVO.getPdItem().getTitle());  
        pdOrderItem.setPrice(itemVO.getPdItem().getPrice());  
        pdOrderItem.setNum(itemVO.getPdCartItem().getNum());  
  
        payment = payment + itemVO.getPdCartItem().getNum() *  
itemVO.getPdItem().getPrice();  
        pdOrderItemMapper.insert(pdOrderItem);  
    }  
    pdOrder.setPayment(payment);  
    pdOrderMapper.insert(pdOrder);  
    return orderId;  
}
```

手动确认

application.yml

```
spring:
  rabbitmq:
    listener:
      simple:
        acknowledge-mode: manual
```

OrderConsumer

```
package com.pd;

import org.springframework.amqp.core.Message;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import com.pd.pojo.PdOrder;
import com.pd.service.OrderService;
import com.rabbitmq.client.Channel;

@Component
public class OrderConsumer {
    // 收到订单数据后, 会调用订单的业务代码, 把订单保存到数据库
    @Autowired
    private OrderService orderService;

    // 添加该注解后, 会从指定的orderQueue接收消息,
    // 并把数据转为 PdOrder 实例传递到此方法
    @RabbitListener(queues="orderQueue")
    public void save(PdOrder pdOrder, Channel channel, Message message)
    {
        System.out.println("消费者");
        System.out.println(pdOrder.toString());
        try {
            orderService.saveOrder(pdOrder);
            channel.basicAck(message.getMessageProperties().getDeliveryTag(), false);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```