## **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。

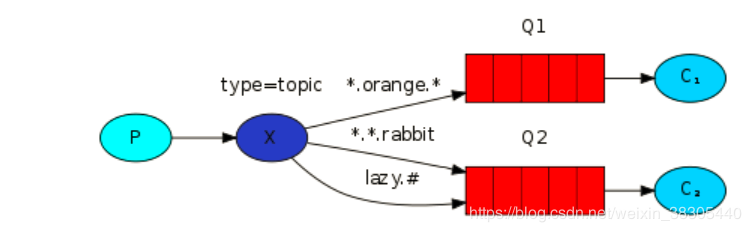
### **主题交换机 Topic exchange**

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 “\*.orange.\*” 绑定。和Q2是 “\*.\*.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);

//自动生成对列名,

//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,

//自动生成对列名,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。

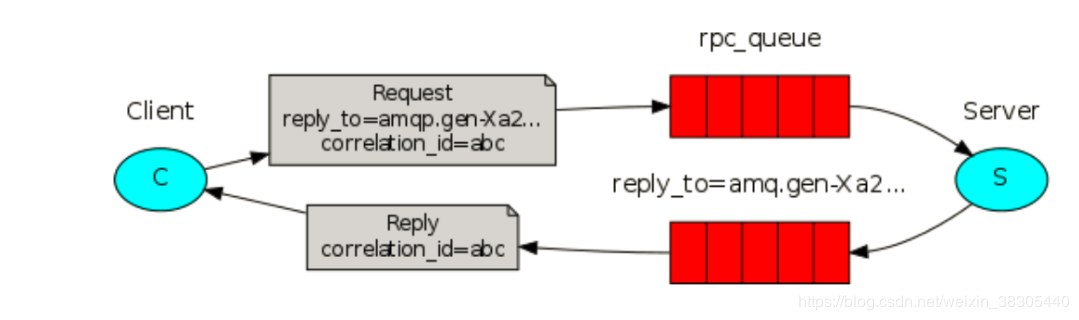
correlationId:将RPC响应与请求关联起来非常有用。

### **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 {

//自动生成对列名,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 {

RPCClient client = new RPCClient();

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);

}

}

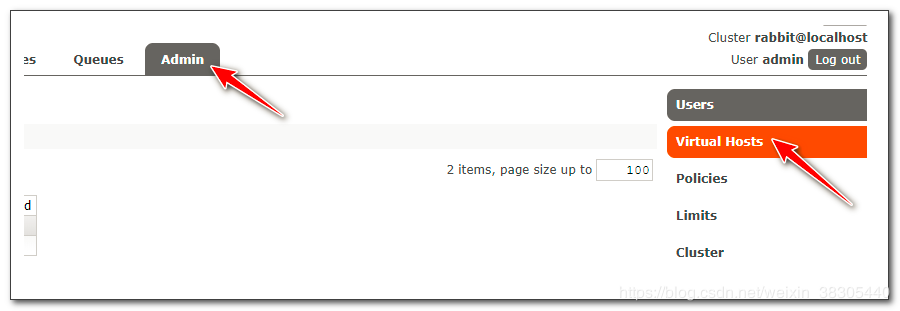
}

# **virtual host**

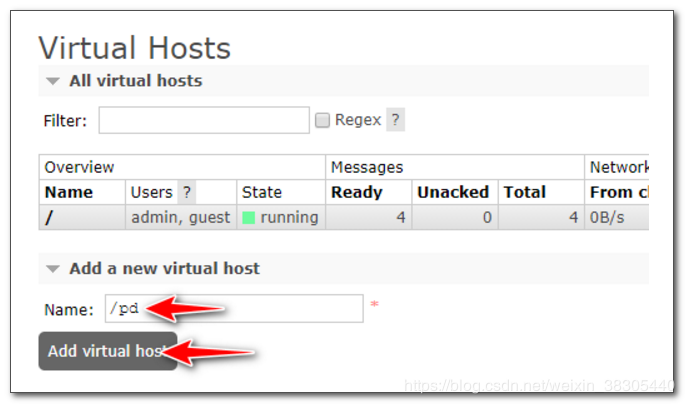
在RabbitMQ中叫做虚拟消息服务器VirtualHost，每个VirtualHost相当于一个相对独立的RabbitMQ服务器，每个VirtualHost之间是相互隔离的。exchange、queue、message不能互通

## **创建virtual host: /pd**

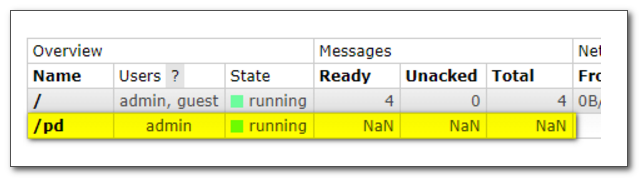
进入虚拟机管理界面



* 添加新的虚拟机’/pd’,名称必须以"/"开头

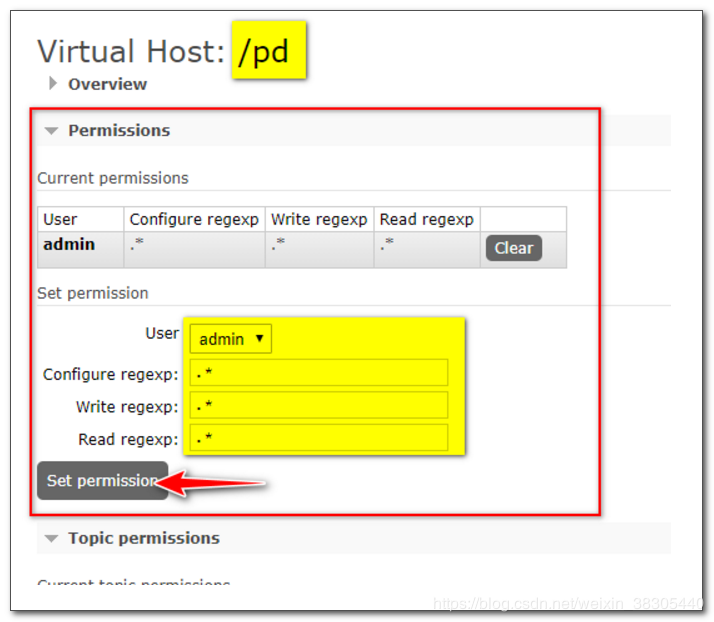


查看添加的结果



## **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

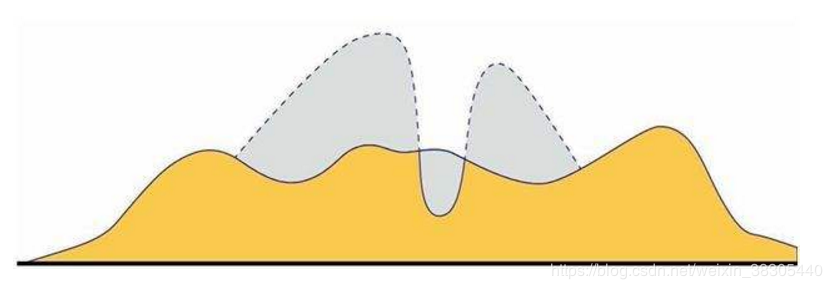


# **拼多商城整合 rabbitmq**

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

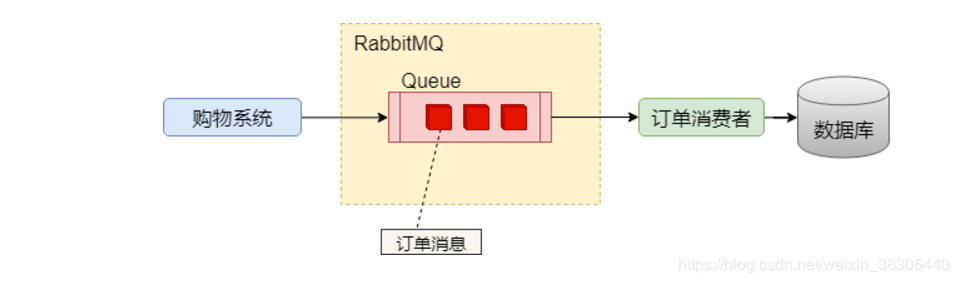
当系统流量突然激增,大量的订单压力,会拖慢业务系统和数据库系统

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

AmqpTemplate 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(10D);

// 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);

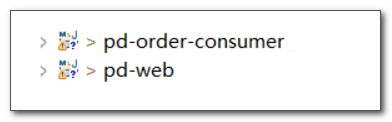
// pdOrderMapper.insert(pdOrder);

// 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**



### **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(10D);

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();

}

}

}

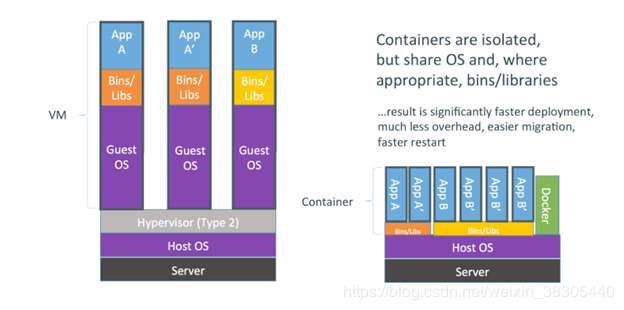
# **docker**

官网是这样介绍docker的：

Docker is an open platform for developers and sysadmins to build, ship, and run distributed applications…

其实看完这句话还是不明白docker究竟是什么

我们可以把他想象成是一个用了一种新颖方式实现的超轻量虚拟机。当然在实现的原理和应用上还是和VM有巨大差别的，并且专业的叫法是应用容器（Application Container）。



比如现在想用MySQL,那就找个装好并配置好的MySQL的容器(可以认为是特殊的,轻量级的虚拟机),运行起来,那么就可以使用 MySQL了。

那么为什么不直接在操作系统中安装一个mysql,而是用容器呢?

安装MySql过程并不简单,要配置安装源,安装依赖包,对mysql进行配置…如果要在多台主机上安装,每台主机都要进行这些繁琐的操作,万一服务器挂了,这一系列操作还要再重来一遍

但有了docker,一个安装配置好的mysql容器,可以直接拿到另一台主机上启动,而不必重新安装mysql

另外,docker还有一重要的用处,就是可以保证开发,测试和生产环境的一致.

# **docker 手册**

中文免费手册 [Docker — 从入门到实践]  
<https://yeasy.gitbooks.io/docker_practice/>

docker 从入门到实践，离线版

docker pull dockerpracticecn/docker\_practice

docker run -it --rm -p 4000:80 dockerpracticecn/docker\_practice

# **centos7 联网安装docker**

### **官方安装手册**

<https://docs.docker.com/install/linux/docker-ce/centos/>

****卸载旧版****

sudo yum remove docker \

docker-client \

docker-client-latest \

docker-common \

docker-latest \

docker-latest-logrotate \

docker-logrotate \

docker-engine

****安装一组工具****

sudo yum install -y yum-utils \

device-mapper-persistent-data \

lvm2

****设置 yum 仓库地址****

sudo yum-config-manager \

--add-repo \

https://download.docker.com/linux/centos/docker-ce.repo

sudo yum-config-manager \

--add-repo \

http://mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.repo

****更新 yum 缓存****

sudo yum makecache fast

****安装新版 docker****

sudo yum install docker-ce docker-ce-cli containerd.io

****启动 docker****

sudo systemctl start docker

****重启 docker****

sudo systemctl restart docker

****设置 docker 开机启动****

sudo systemctl enable docker

****运行 hello-world 镜像，验证 docker****

sudo docker run hello-world

### **镜像加速**

由于国内网络问题，需要配置加速器来加速。  
修改配置文件 /etc/docker/daemon.json

vim /etc/docker/daemon.json

****添加以下内容****

{

"registry-mirrors": ["http://hub-mirror.c.163.com"]

}

****之后重新启动服务。****

sudo systemctl daemon-reload

sudo systemctl restart docker

# **基本概念**

## **镜像**

Docker 镜像是一个特殊的文件系统，除了提供容器运行时所需的程序、库、资源、配置等文件外，还包含了一些为运行时准备的一些配置参数（如匿名卷、环境变量、用户等）。镜像不包含任何动态数据，其内容在构建之后也不会被改变。

镜像只是一个虚拟的概念，其实际体现并非由一个文件组成，而是由一组文件系统组成，或者说，由多层文件系统联合组成。

镜像构建时，会一层层构建，前一层是后一层的基础。每一层构建完就不会再发生改变，后一层上的任何改变只发生在自己这一层。比如，删除前一层文件的操作，实际不是真的删除前一层的文件，而是仅在当前层标记为该文件已删除。在最终容器运行的时候，虽然不会看到这个文件，但是实际上该文件会一直跟随镜像。因此，在构建镜像的时候，需要额外小心，每一层尽量只包含该层需要添加的东西，任何额外的东西应该在该层构建结束前清理掉。

分层存储的特征还使得镜像的复用、定制变的更为容易。甚至可以用之前构建好的镜像作为基础层，然后进一步添加新的层，以定制自己所需的内容，构建新的镜像。