**[SpringCloud之RabbitMQ消息队列原理及配置](https://www.cnblogs.com/jing99/p/11679426.html)**

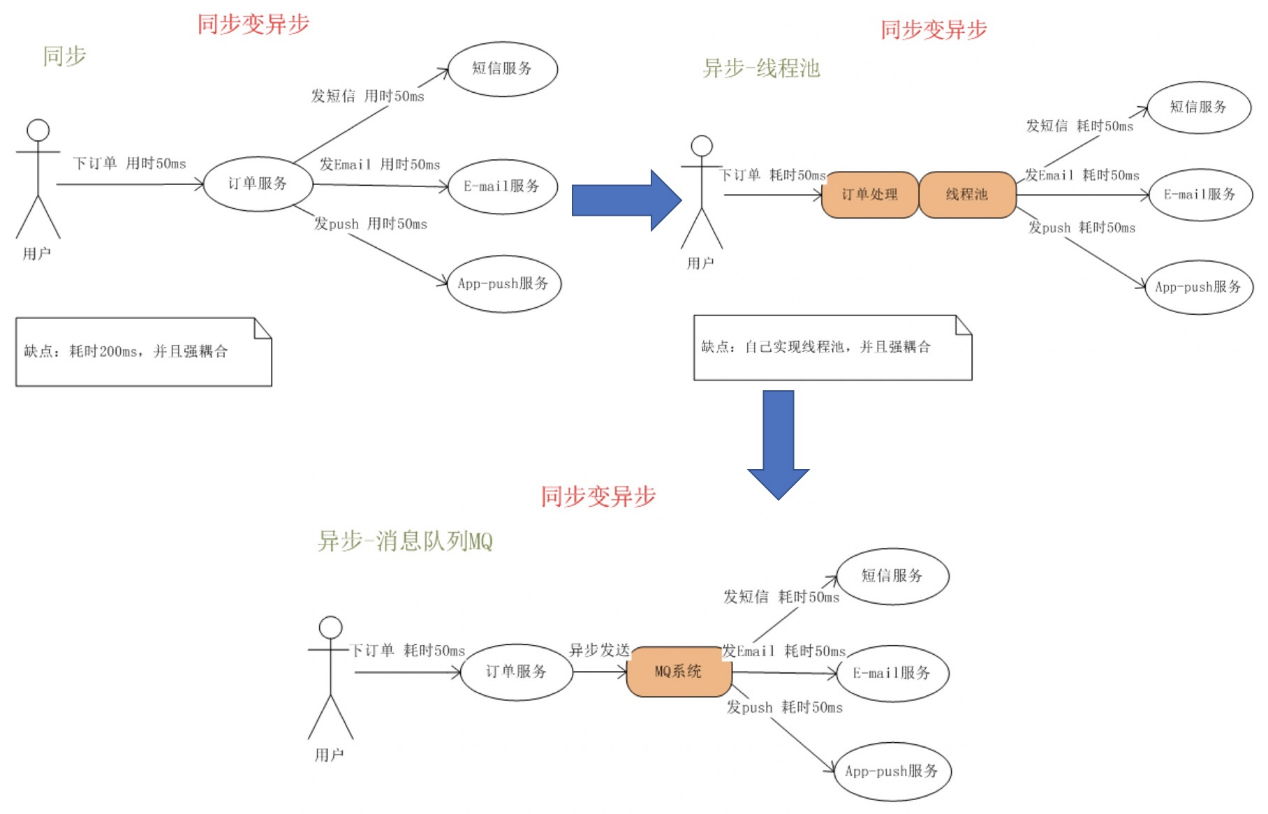
　　本篇章讲解RabbitMQ的用途、原理以及配置，RabbitMQ的安装请查看***[SpringCloud之RabbitMQ安装](https://www.cnblogs.com/jing99/p/11673987.html)***

**一、MQ用途**

**1、同步变异步消息**

　　场景：用户下单完成后，发送邮件和短信通知。

　　运用消息队列之后，用户下单完之后，下单信息写入数据库，再写入消息队列，发送邮件和发送短信各自去消息队列进行读取，节省时间，提高效率。



**2、应用解耦**

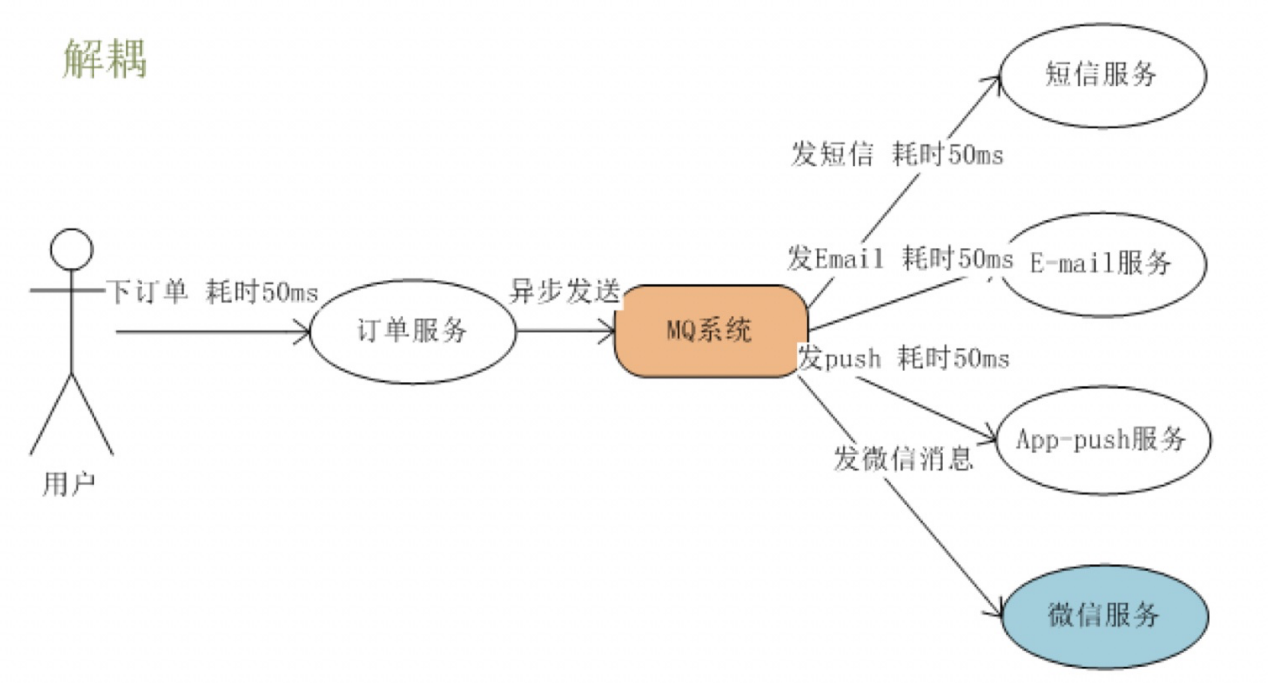
　　场景：用户下单后，订单系统需要多渠道通知用户。

　　下单服务系统：用户使用下单服务后，将下单信息写入数据库，下单成功。

　　短信服务系统：用户下单后，将短信信息写入消息队列，以发送短信信息通知用户交易信息。

　　邮件服务系统：用户下单后，将邮件信息写入消息队列，以发送邮件信息通知用户交易信息。

　　这样，如果微信通知不能正常使用，也不影响用户下单，用户下单后，只用把下单通知信息写入消息队列，不用关心后续操作，实现了订单系统和通知系统的解耦。



**3、流量削峰**

　　一般在秒杀或者团购活动中使用。

　　场景：秒杀活动，一般会因为流量过大，导致流量暴增，应用挂掉。针对这个问题，一般需要在应用前端加入消息队列。

　　　　a.可以控制活动的人数

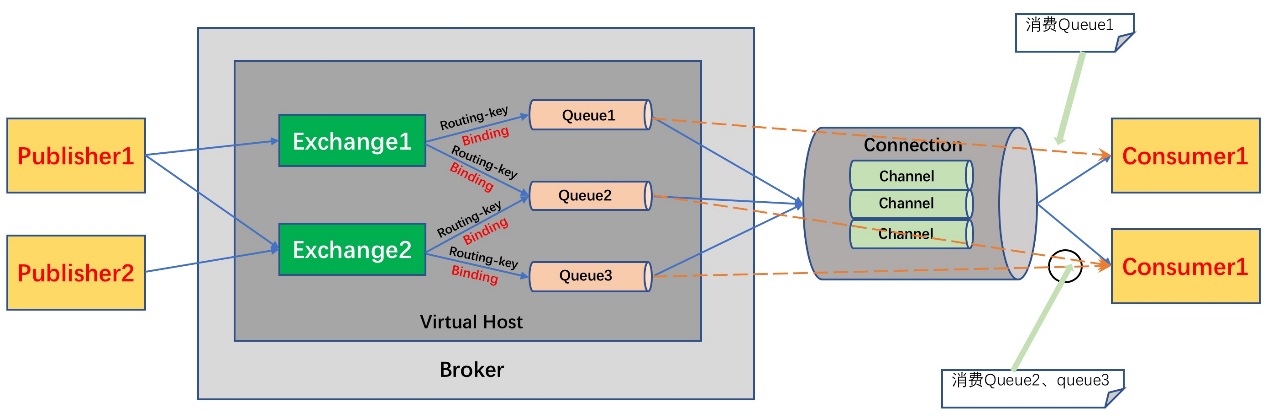
　　　　b.可以缓解短时间内高流量压垮应用

　　用户的请求，服务器接收后，首先写入消息队列，如果消息队列的数量大于最大的数量，则直接抛弃用户请求或者跳转错误页面。

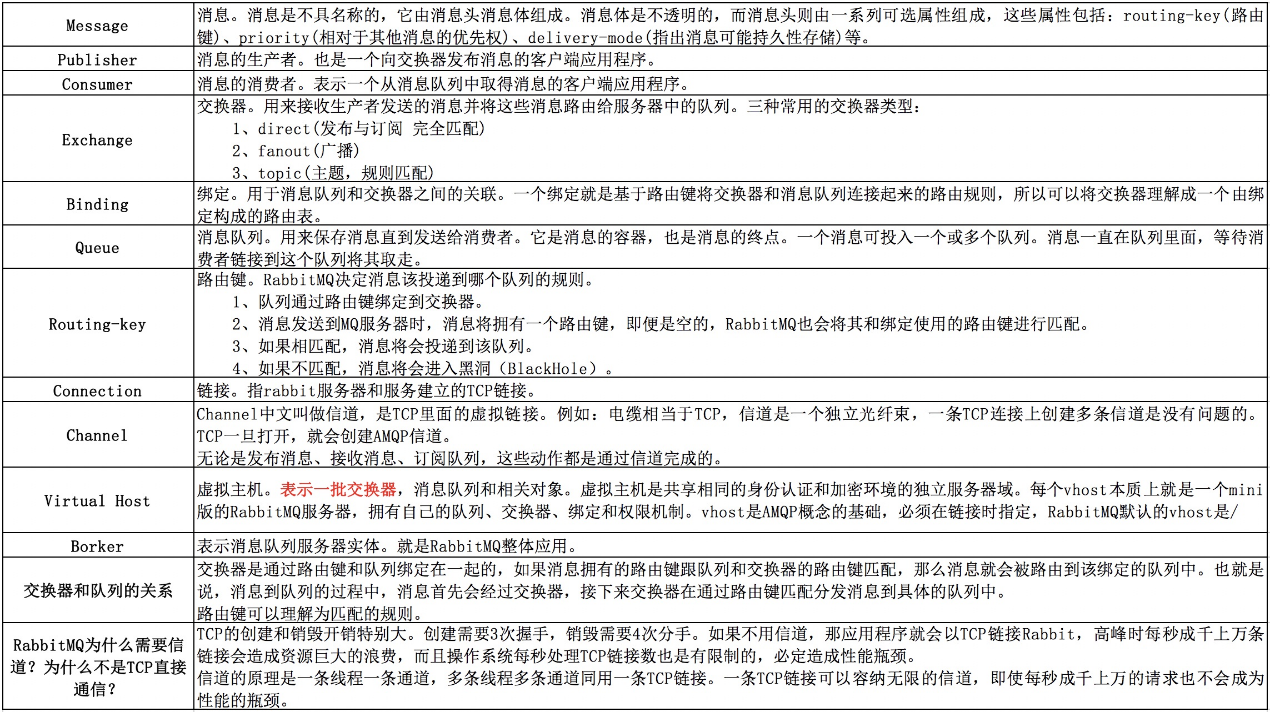


**二、RabbitMQ原理介绍**

　　如图所示：



　　各组件意义如下：



**三、RabbitMQ应用**

　　RabbitMQ包依赖（**spring-boot-starter-amqp**）：

[复制代码](javascript:void(0);)

<!-- rabbitMQ的依赖。rabbitmq已经被spring-boot做了整合访问实现。

spring cloud也对springboot做了整合逻辑。所以rabbitmq的依赖可以在spring cloud中直接使用。

-->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>**spring-boot-starter-amqp**</artifactId>

</dependency>

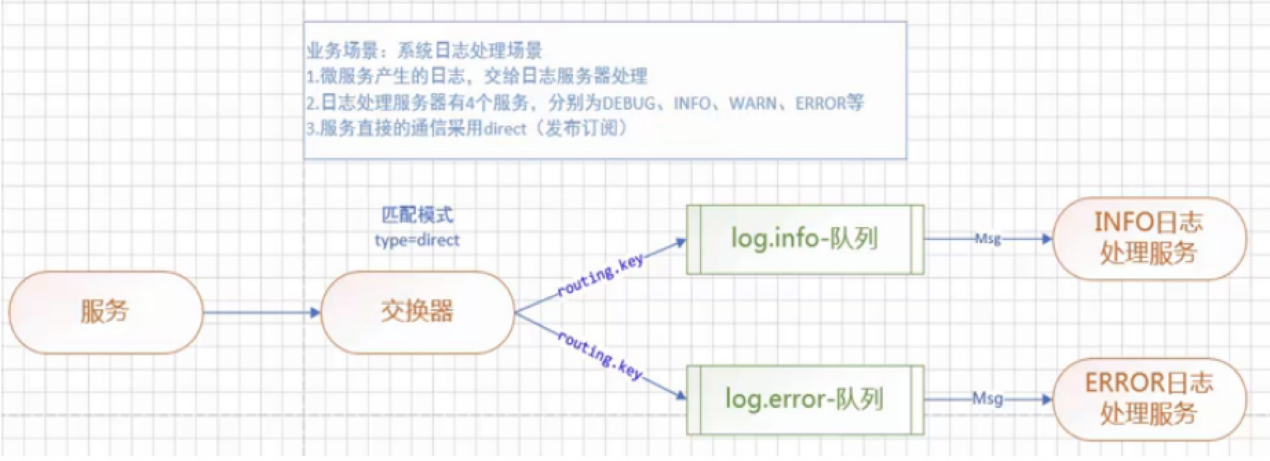
[复制代码](javascript:void(0);)

**1、Direct交换器**

　　是一种点对点，实现发布/订阅标准的交换器。Producer发送消息到RabbitMQ中，MQ中的Direct交换器接受到消息后，会根据Routing Key来决定这个消息要发送到哪一个队列中。Consumer则负责注册一个队列监听器，来监听队列的状态，当队列状态发生变化时，消费消息。注册队列监听需要提供交换器信息，队列信息和路由键信息。

　　这种交换器通常用于点对点消息传输的业务模型中。如电子邮箱。

　　如下图所示日志处理MQ示例：



　　Producer全局配置文件：

[复制代码](javascript:void(0);)

spring.application.name=direct-producer

server.port=8082

# 必要配置

# 配置rabbitmq链接相关信息。key都是固定的。是springboot要求的。

# rabbitmq安装位置

spring.rabbitmq.host=localhost

# rabbitmq的端口

spring.rabbitmq.port=5672

# rabbitmq的用户名

spring.rabbitmq.username=test

# rabbitmq的用户密码

spring.rabbitmq.password=123456

# 可选配置

# **配置producer中操作的Queue和Exchange相关信息的。key是自定义的。为了避免硬编码（代码中可以写死）。**

# exchange的命名。交换器名称可以随意定义。

mq.config.exchange=log.direct

# 路由键， 是定义某一个路由键。 info级别日志使用的queue的路由键。

mq.config.queue.info.routing.key=log.info.routing.key

# 路由键，error级别日志使用的queue的路由键。

mq.config.queue.error.routing.key=log.error.routing.key

[复制代码](javascript:void(0);)

　　Producer消息发送类：

[复制代码](javascript:void(0);)

/\*\*

\* 消息发送者 - Producer。

\* @Component **Producer类型的对象，必须交由Spring容器管理。**

\* **使用SpringBoot提供的AMQP启动器，来访问rabbitmq的时候，都是通过AmqpTemplate来实现的。**

\* 如果全局配置文件中，配置了rabbitmq相关内容，且工程依赖了starter-amqp，则**spring容器自动创建AmqpTemplate对象。**

**\*/**

@Component

public class Sender {

@Autowired

private AmqpTemplate rabbitAmqpTemplate;

//exchange 交换器名称

@Value("${mq.config.exchange}")

private String exchange;

//routingkey 路由键

@Value("${mq.config.queue.info.routing.key}")

private String routingkey;

/\*

\* 发送消息的方法

\*/

public void send(LogMessage msg){

/\*\*

\* convertAndSend - 转换并发送消息的template方法。

\* 是将传入的普通java对象，转换为rabbitmq中需要的message类型对象，并发送消息到rabbitmq中。

\* 参数一：**交换器名称。 类型是String**

\* 参数二：**路由键。 类型是String**

\* 参数三：**消息，是要发送的消息内容对象。类型是Object**

\*/

this.rabbitAmqpTemplate.convertAndSend(this.exchange, this.routingkey, msg);

}

}

[复制代码](javascript:void(0);)

　　Producer实体类：

[复制代码](javascript:void(0);)

/\*\*

\* **消息内容载体，在rabbitmq中，存储的消息可以是任意的java类型的对象。**

\* **强制要求，作为消息数据载体的类型，必须是Serializable的。**

\* **如果消息数据载体类型未实现Serializable，在收发消息的时候，都会有异常发生。**

\*/

public class LogMessage implements Serializable {

private Long id;

private String msg;

private String logLevel;

private String serviceType;

private Date createTime;

private Long userId;

public LogMessage() {

super();

}

public LogMessage(Long id, String msg, String logLevel, String serviceType, Date createTime, Long userId) {

super();

this.id = id;

this.msg = msg;

this.logLevel = logLevel;

this.serviceType = serviceType;

this.createTime = createTime;

this.userId = userId;

}

@Override

public String toString() {

return "LogMessage [id=" + id + ", msg=" + msg + ", logLevel=" + logLevel + ", serviceType=" + serviceType

+ ", createTime=" + createTime + ", userId=" + userId + "]";

}

public Long getId() {

return id;

}

public void setId(Long id) {

this.id = id;

}

public String getMsg() {

return msg;

}

public void setMsg(String msg) {

this.msg = msg;

}

public String getLogLevel() {

return logLevel;

}

public void setLogLevel(String logLevel) {

this.logLevel = logLevel;

}

public String getServiceType() {

return serviceType;

}

public void setServiceType(String serviceType) {

this.serviceType = serviceType;

}

public Date getCreateTime() {

return createTime;

}

public void setCreateTime(Date createTime) {

this.createTime = createTime;

}

public Long getUserId() {

return userId;

}

public void setUserId(Long userId) {

this.userId = userId;

}

}

[复制代码](javascript:void(0);)

　　Producer消息产生测试类：

[复制代码](javascript:void(0);)

/\*\*

\* Direct交换器

\* Producer测试。

\* 注意：

\* **在rabbitmq中，consumer都是listener监听模式消费消息的。**

\* 一般来说，**在开发的时候，都是先启动consumer，确定有什么exchange、queue、routing-key，然后再启动producer。**

\* 然后再启动producer发送消息，。

\*/

@RunWith(SpringRunner.class)

@SpringBootTest(classes=SpringbootServerApplication.class)

public class QueueTest {

@Autowired

private Sender sender;

/\*

\* 测试消息队列

\*/

@Test

public void testSend()throws Exception{

Long id = 1L;

while(true){

Thread.sleep(1000);

**this.sender.send(**new LogMessage(id,"test log", "info", "订单服务", new Date(), id));

id++;

}

}

}

[复制代码](javascript:void(0);)

　　Consumer全局配置：

[复制代码](javascript:void(0);)

spring.application.name=direct-consumer

server.port=8083

spring.rabbitmq.host=localhost

spring.rabbitmq.port=5672

spring.rabbitmq.username=test

spring.rabbitmq.password=123456

# 自定义配置。 **配置交换器exchange、路由键routing-key、队列名称 queue name；在RabbitMQ中队列的生成**

# 交换器名称

mq.config.exchange=log.direct

# info级别queue的名称

mq.config.queue.info=log.info

# info级别的路由键

mq.config.queue.info.routing.key=log.info.routing.key

# error级别queue的名称

mq.config.queue.error=log.error

# error级别的路由键

mq.config.queue.error.routing.key=log.error.routing.key

[复制代码](javascript:void(0);)

　　Consumer消费者：

[复制代码](javascript:void(0);)

/\*\*

\* 消息接收者 - consumer

\*

\* **@RabbitListener - 可以注解类和方法。**

\* 注解类，当表当前类的对象是一个rabbit listener。

\* 监听逻辑明确，可以由更好的方法定义规范。

\* 必须配合@RabbitHandler才能实现rabbit消息消费能力，**一个类可以有多个方法，但是仅有一个方法注解@RabbitHandler。**

**\* 注解方法，代表当前方法是一个rabbit listener处理逻辑。**

**\* 方便开发，一个类中可以定义若干个listener逻辑。**

\* 方法定义规范可能不合理。如：一个方法的处理逻辑太多，造成方法的bad smell。

\*

\* @RabbitListener - 代表当前类型是一个rabbitmq的监听器。

\*  **bindings:绑定队列**

\* @QueueBinding - @RabbitListener.bindings属性的类型。绑定一个队列。

**\* value:绑定队列， Queue类型。**

**\* exchange:配置交换器， Exchange类型。**

**\* key:路由键，字符串类型。**

\*

\* @Queue - 队列。

**\* value:队列名称**

**\* autoDelete:是否是一个临时队列。**

**\* true ：当所有的consumer关闭后，自动删除queue。**

**\* false：当任意一个consumer启动并创建queue后，如果queue中有消息未消费，无论是否有consumer继续执行，都保存queue。**

\*

\* @Exchange - 交换器

**\* value:为交换器起个名称**

**\* type:指定具体的交换器类型**

\*/

@Component

@RabbitListener(

**bindings**=@**QueueBinding**(

**value**=@Queue(value="${mq.config.queue.error}",autoDelete="false"),

**exchange**=@Exchange(value="${mq.config.exchange}",**type=ExchangeTypes.DIRECT**),

**key**="${mq.config.queue.error.routing.key}"

)

)

public class ErrorReceiver {

/\*\*

\* 消费消息的方法。**采用消息队列监听机制**

\* @RabbitHandler - 代表当前方法是监听队列状态的方法，就是队列状态发生变化后，执行的消费消息的方法。

\* 方法参数。就是处理的消息的数据载体类型。

\*/

@RabbitHandler

public void process(LogMessage msg){

System.out.println("Error..........receiver: "+msg);

}

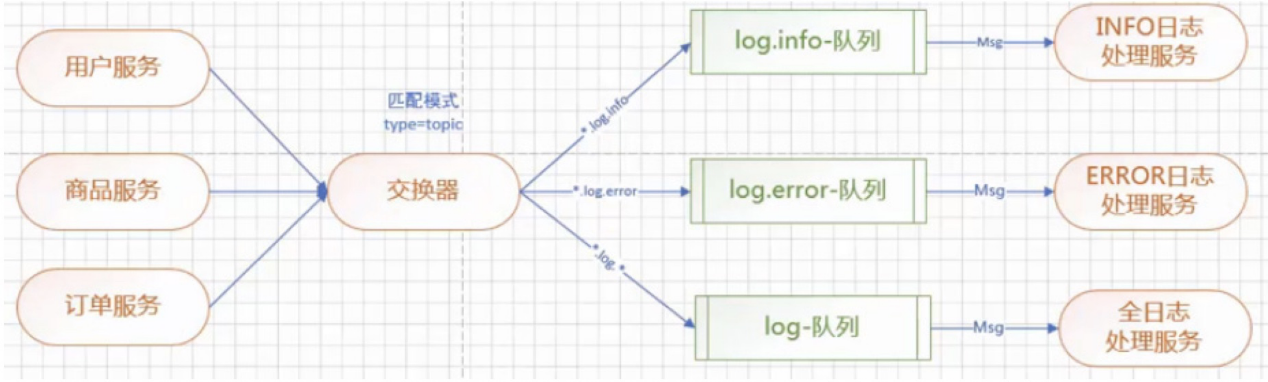
}

[复制代码](javascript:void(0);)

**2、Topic交换器**

　　主题交换器，也称为规则匹配交换器。是通过自定义的模糊匹配规则来决定消息存储在哪些队列中。当Producer发送消息到RabbitMQ中时，MQ中的交换器会根据路由键来决定消息应该发送到哪些队列中。Consumer同样是注册一个监听器到队列，监听队列状态，当队列状态发生变化时，消费消息。注册监听器需要提供交换器信息，队列信息和路由键信息。

　　如下图所示日志处理MQ示例：



　　Producer公共配置文件：

[复制代码](javascript:void(0);)

spring.application.name=topic-producer

spring.rabbitmq.host=192.168.1.122

spring.rabbitmq.port=5672

spring.rabbitmq.username=test

spring.rabbitmq.password=123456

mq.config.exchange=log.topic

[复制代码](javascript:void(0);)

　　Producer的User实体日志发送类：

[复制代码](javascript:void(0);)

/\*\*

\* 消息发送者

\*/

@Component

public class UserSender {

@Autowired

private AmqpTemplate rabbitAmqpTemplate;

//exchange 交换器名称

@Value("${mq.config.exchange}")

private String exchange;

/\*

\* 发送消息的方法

\*/

public void send(String msg){

//向消息队列发送消息

//参数一：交换器名称。

//参数二：路由键

//参数三：消息

this.rabbitAmqpTemplate.convertAndSend(this.exchange,"user.log.debug", "user.log.debug....."+msg);

this.rabbitAmqpTemplate.convertAndSend(this.exchange,"user.log.info", "user.log.info....."+msg);

this.rabbitAmqpTemplate.convertAndSend(this.exchange,"user.log.warn","user.log.warn....."+msg);

this.rabbitAmqpTemplate.convertAndSend(this.exchange,"user.log.error", "user.log.error....."+msg);

}

}

[复制代码](javascript:void(0);)

　　Producer的Order实体日志发送类：

[复制代码](javascript:void(0);)

/\*\*

\* 消息发送者

\*/

@Component

public class OrderSender {

@Autowired

private AmqpTemplate rabbitAmqpTemplate;

//exchange 交换器名称

@Value("${mq.config.exchange}")

private String exchange;

/\*

\* 发送消息的方法

\*/

public void send(String msg){

//向消息队列发送消息

//参数一：交换器名称。

//参数二：路由键

//参数三：消息

this.rabbitAmqpTemplate.convertAndSend(this.exchange,"order.log.debug", "order.log.debug....."+msg);

this.rabbitAmqpTemplate.convertAndSend(this.exchange,"order.log.info", "order.log.info....."+msg);

this.rabbitAmqpTemplate.convertAndSend(this.exchange,"order.log.warn","order.log.warn....."+msg);

this.rabbitAmqpTemplate.convertAndSend(this.exchange,"order.log.error", "order.log.error....."+msg);

}

}

[复制代码](javascript:void(0);)

　　Producer测试类：

[复制代码](javascript:void(0);)

/\*\*

\* 消息队列测试类

\* @author Administrator

\*

\*/

@RunWith(SpringRunner.class)

@SpringBootTest(classes=SpringbootServerApplication.class)

public class QueueTest {

@Autowired

private UserSender usersender;

@Autowired

private ProductSender productsender;

@Autowired

private OrderSender ordersender;

/\*

\* 测试消息队列

\*/

@Test

public void test() throws InterruptedException{

while(true){

Thread.sleep(1000);

this.usersender.send("UserSender.....");this.ordersender.send("OrderSender......");

}

}

}

[复制代码](javascript:void(0);)

　　可以看出Producer的发送和Direct没有区别，Consumer的全局配置文件：

[复制代码](javascript:void(0);)

spring.application.name=topic-consumer

spring.rabbitmq.host=192.168.1.122

spring.rabbitmq.port=5672

spring.rabbitmq.username=test

spring.rabbitmq.password=123456

mq.config.exchange=log.topic

mq.config.queue.info=log.info

mq.config.queue.error=log.error

mq.config.queue.logs=log.all

[复制代码](javascript:void(0);)

　　Consumer中的info日志消费者：

[复制代码](javascript:void(0);)

@Component

@RabbitListener(

bindings=@QueueBinding(

value=@Queue(value="${mq.config.queue.info}",autoDelete="true"),

exchange=@Exchange(value="${mq.config.exchange}",**type=ExchangeTypes.TOPIC**),

**key="\*.log.info"**

)

)

public class InfoReceiver {

@RabbitHandler

public void process(String msg){

System.out.println("......Info........receiver: "+msg);

}

}

[复制代码](javascript:void(0);)

　　Consumer中的全体日志消费者：

[复制代码](javascript:void(0);)

/\*\*

\* **和direct交换器的区别是： Exchange的类型为TOPIC。**

\* 全日志处理。

\*/

@Component

@RabbitListener(

bindings=@QueueBinding(

value=@Queue(value="${mq.config.queue.logs}",autoDelete="true"),

exchange=@Exchange(value="${mq.config.exchange}",**type=ExchangeTypes.TOPIC**),

**key="\*.log.\*"**

)

)

public class LogsReceiver {

@RabbitHandler

public void process(String msg){

System.out.println("......All........receiver: "+msg);

}

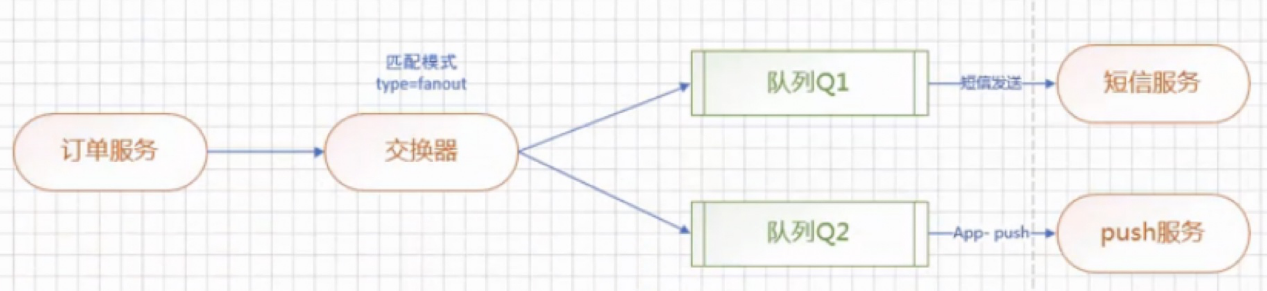
}

[复制代码](javascript:void(0);)

**3、Fanout交换器**

　　广播交换器。这种交换器会将接收到的消息发送给绑定的所有队列中。当Producer发送消息到RabbitMQ时，交换器会将消息发送到已绑定的所有队列中，这个过程交换器不会尝试匹配路由键，所以消息中不需要提供路由键信息。Consumer仍旧注册监听器到队列，监听队列状态，当队列状态发生变化，消费消息。注册监听器需要提供交换器信息和队列信息。

　　如下图所示短信、APP推送的MQ示例：



　　由于Producer的测试类和以上无差别，不再赘述，如下Producer的发送类：

[复制代码](javascript:void(0);)

/\*\*

\* 消息发送者

\* fanout交换器 -

\* **使用fanout交换器的时候，交换器是忽略routing-key的匹配。**

\* **因为广播不需要考虑路由键的匹配，只考虑在Exchange上绑定了多少个queue，这个由Consumer的配置决定。**

\* **会将接受到的消息发送到所有的绑定的queue中，进行消息的缓存。**

\*/

@Component

public class Sender {

@Autowired

private AmqpTemplate rabbitAmqpTemplate;

//exchange 交换器名称

@Value("${mq.config.exchange}")

private String exchange;

/\*

\* 发送消息的方法

\*/

public void send(String msg){

//向消息队列发送消息

//参数一：交换器名称。

//**参数二：路由键　　无需填写，填写了也无效**

//参数三：消息

this.rabbitAmqpTemplate.convertAndSend(this.exchange,**"A"**, msg);

}

}

[复制代码](javascript:void(0);)

　　如下所示Consumer的SMS消费类：

[复制代码](javascript:void(0);)

/\*\*

\* **使用fanout交换器的时候，可以在consumer中省略routing-key的配置。**

**\* 因为fanout交换器忽略routing-key的匹配，即使配置当type=ExchangeTypes.FANOUT时也无效。**

\*/

@Component

@RabbitListener(

bindings=@QueueBinding(

value=@Queue(value="${**mq.config.queue.sms**}",autoDelete="true"),

exchange=@Exchange(value="${mq.config.exchange}",**type=ExchangeTypes.FANOUT**)

)

)

public class SmsReceiver {

@RabbitHandler

public void process(String msg){

System.out.println("Sms........receiver: "+msg);

}

}

[复制代码](javascript:void(0);)

　　如Consumer的Publish消费类：

[复制代码](javascript:void(0);)

@Component

@RabbitListener(

bindings=@QueueBinding(

value=@Queue(value="${**mq.config.queue.push**}",autoDelete="true"),

exchange=@Exchange(value="${mq.config.exchange}",type=ExchangeTypes.FANOUT)

)

)

public class PushReceiver {

@RabbitHandler

public void process(String msg){

System.out.println("Push..........receiver: "+msg);

}

}

[复制代码](javascript:void(0);)

**四、RabbitMQ消息可靠性处理**

　　前面内容，如果consumer未启动，而producer发送了消息。则消息会丢失。如果consumer先启动，创建queue后，producer发送消息可以正常消费。那么当所有的consumer宕机的时候，queue会auto-delete，消息仍旧会丢失。这种情况，消息不可靠。有丢失的可能。

　　Rabbitmq的消息可靠性处理，分为两部分。

* 消息不丢失。当consumer全部宕机后，消息不能丢失。 ------持久化解决
* 消息不会错误消费。当consumer获取消息后，万一consumer在消费消息的过程中发生了异常，如果rabbitmq一旦发送消息给consumer后立刻删除消息，也会有消息丢失的可能。 -------确认机制解决

**1、消息持久化**

* @Queue注解中的属性 - autoDelete：当所有消费客户端连接断开后，是否自动删除队列 。true：删除   false：不删除
* @Exchange注解中的属性 - autoDelete：当交换器所有的绑定队列都不再使用时，是否自动删除交换器（更粗粒度，不建议）。true：删除   false：不删除

**2、消息确认机制 ACK - acknowledge**

　　什么是消息确认机制？

　　如果在消息处理过程中，消费者的服务器在处理消息时发生异常，那么这条正在处理的消息就很可能没有完成消息的消费，如果RabbitMQ在Consumer消费消息后立刻删除消息，则可能造成数据丢失。为了保证数据的可靠性，RabbitMQ引入了消息确认机制。

* 消息确认机制是消费者Consumer从RabbitMQ中收到消息并处理完成后，反馈给RabbitMQ的，当RabbitMQ收到确认反馈后才会将此消息从队列中删除。
* 如果某Consumer在处理消息时出现了网络不稳定，服务器异常等现象时，那么就不会有消息确认反馈，RabbitMQ会认为这个消息没有正常消费，会将消息重新放入队列中。
* 如果在Consumer集群环境下，RabbitMQ未接收到Consumer的确认消息时，会立即将这个消息推送给集群中的其他Consumer，保证不丢失消息。
* 如果Consumer没有确认反馈，RabbitMQ将永久保存消息。

　　消息确认机制默认都是开启状态的，同时不推荐关闭消息确认机制。

　　注意：如果Consumer没有处理消息确认，将导致严重后果。如：所有的Consumer都没有正常反馈确认信息，并退出监听状态，消息则会永久保存，并处于锁定状态，直到消息被正常消费为止。消息的发送者Producer如果持续发送消息到RabbitMQ，那么消息将会堆积，持续占用RabbitMQ所在服务器的内存，导致“内存泄漏”问题。

　　消息确认机制处理方案：

**编码异常处理（推荐）**

　　通过编码处理异常的方式，保证消息确认机制正常执行。**这种处理方案也可以有效避免消息的重复消费。**

　　异常处理，不是让Consumer编码catch异常后，直接丢弃消息，或反馈ACK确认消息。而是做异常处理的。该抛的异常，还得抛，保证ACK机制的正常执行。或者使用其他的手法，实现消息的再次处理。如：catch代码块中，将未处理成功的消息，重新发送给MQ。如：catch代码中，本地逻辑的重试（使用定时线程池重复执行任务3次。）

**配置重试次数处理**

　　通常来说，消息重试3次以上未处理成功，就是Consumer开发出现了严重问题。需要修改Consumer代码，提升版本/打补丁之类的处理方案。

　　通过全局配置文件，开启消息消费重试机制，配置重试次数。当RabbitMQ未收到Consumer的确认反馈时，会根据配置来决定重试推送消息的次数，当重试次数使用完毕，无论是否收到确认反馈，RabbitMQ都会删除消息，避免内存泄漏的可能。具体配置如下：

#开启重试

spring.rabbitmq.listener.retry.enabled=true

#重试次数，默认为3次

spring.rabbitmq.listener.retry.max-attempts=5

**五、常用MQ产品对比和选择**

　　社区活跃度：RabbitMQ > ActiveMQ = RocketMQ > kafka

　　消息持久化：RabbitMQ、ActiveMQ、RocketMQ、kafka都支持持久化。ZeroMQ不支持持久化。

　　高并发： RabbitMQ = kafka > RocketMQ > ActiveMQ。RabbitMQ高并发是基于ErLang的。ErLang本身就是针对高并发提供的一种开发脚本语言。

　　吞吐量：RabbitMQ = kafka > RocketMQ > ActiveMQ。小型项目（并发吞吐低于万级别）使用ActiveMQ。中型项目（并发吞吐10万~100万级），可选RocketMQ、ActiveMQ。大型项目优先考虑RabbitMQ和Kafka。

　　综合技术：RabbitMQ和kafka最好。RocketMQ次之。ActiveMQ最弱。如：可靠性、路由、集群、事务、高可用队列、消息可靠排序、持久化、可视化管理工具等。

　　RabbitMQ和Kafka选择：**建议Kafka针对日志处理。其他使用RabbitMQ**。商业项目中，如果现有的系统架构已经使用了某一个MQ产品，且没有业务和性能上的问题，不推荐切换MQ产品。