RabbitMQ

MQ基本概念

MQ全称Message Queue(消息队列)，是在消息的传输过程中保存消息的容器，多用于分布式系统之间进行通信。

- MQ，消息队列，存储消息的中间件
- 分布式系统通信两种方式：直接远程调用和借助第三方完成间接通信。
- 发送方称为生产者，接收方称为消费者。

MQ的优势

优势：

- 应用解耦
- 异步提速
- 削峰填谷

劣势：

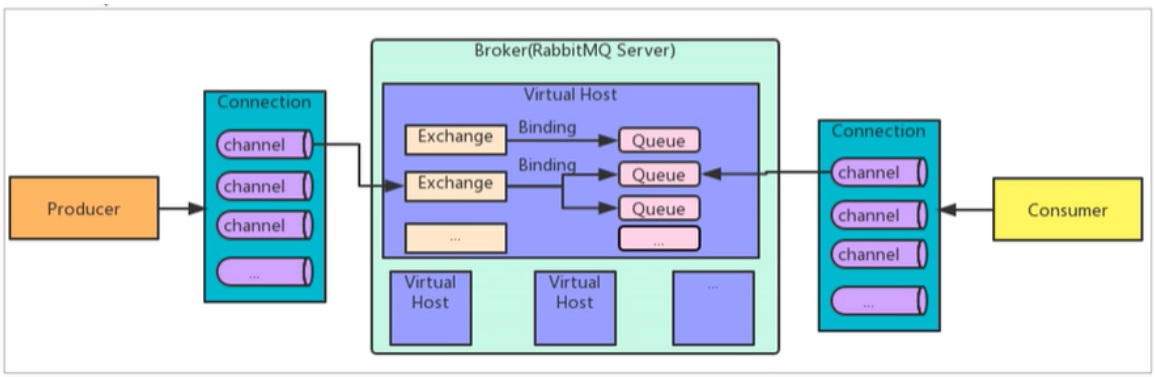
- 系统可用性降低
- 系统复杂度提高
- 一致性问题

RabbitMQ简介

AMQP，即 Advanced Message Queuing Protocol (高级消息队列协议），是一个网络协议，是应用层协议的一个开放标准，为面向消息的中间件设计。基于此协议的客户端与消息中间件可传递消息，并不受客户端/中间件不同产品，不同的开发语言等条件的限制。2006年，AMQP规范发布。类比HTTP。

2007年，Rabbit 技术公司基于AMQP标准开发的RabbitMQ 1.0发布。RabbitMQ采用Erlang 语言开发。Erlang 语言由Ericson设计，专门为开发高并发和分布式系统的一种语言，在电信领域使用广泛。

RabbitMQ基础架构如下图：



相关概念：

Broker:接收和分发消息的应用，RabbitMQ Server就是 Message Broker

Virtual host: 出于多租户和安全因素设计的，把AMQP的基本组件划分到一个虚拟的分组中，类似于网络中的namespace概念。当多个不同的用户使用同一个RabbitMQ server提供的服务时，可以划分出多个vhost，每个用户在自己的 vhost创建exchange / queue等

Connection: publisher / consumer和 broker 之间的TCP连接

Channel:如果每一次访问RabbitMQ都建立一个Connection，在消息量大的时候建立TCP Connection的开销将是巨大的，效率也较低。Channel是在 connection内部建立的逻辑连接，如果应用程序支持多线程，通常每个thread创建单独的channel进行通讯，AMQP method包含了channel id帮助客户端和message broker 识别channel，所以channel之间是完全隔离的。Channel作为轻量级的Connection极大减少了操作系统建立TCP connection的开销

Exchange: message到达 broker的第一站，根据分发规则，匹配查询表中的 routing key，分发消息到queue 中去。常用的类型有: direct (point-to-point), topic(publish-subscribe) and fanout(multicast)

Queue:消息最终被送到这里等待consumer取走

Binding: exchange和queue之间的虚拟连接，binding 中可以包含 routing key。Binding 信息被保存到exchange 中的查询表中，用于message 的分发依据

`RabbitMQ提供了6种工作模式`:简单模式、work queues、Publish/Subscribe 发布与订阅模式、Routing 路由模式、Topics主题模式、RPC远程调用模式(远程调用，不太算MQ;暂不作介绍)。

JMS

JMS即Java消息服务（JavaMessage Service）应用程序接口，是一个Java平台中关于面向消息中间件的API。

JMS是javaEE规范中的一种，类比JDBC。

很多消息中间件都实现了JMS规范，例如：ActiveMQ。RabbitMQ官方没有提供JMS的实现包，但是开源社区有。

RabbitMQ的安装和配置

1、安装Erlang环境支持

安装之前要安装一些必要的库：

apt-get install build-essential

apt-get install libncurses5-dev

apt-get install libssl-dev

apt-get install m4

apt-get install unixodbc unixodbc-dev

apt-get install freeglut3-dev libwxgtk2.8-dev

apt-get install xsltproc

apt-get install fop

apt-get install tk8.5

安装好后安装Erlang:

```
# apt-get install erlang
```

查看Erlang版本：

```
erl --version
```

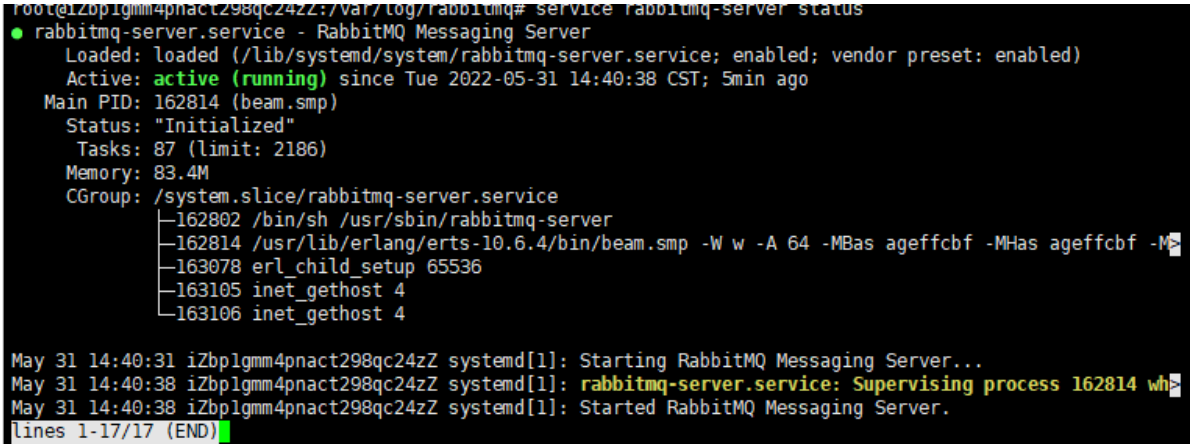
2、安装RabbitMQ

直接安装

```
apt-get install rabbitmq-server
```

查看运行状态

```
service rabbitmq-server status
```



3、配置RabbitMQ

查看log文件：

```
cd /var/log/rabbitmq/
```

打开log文件：

```
vim rabbit@begoit-916-01.log
```

看到在config file(s)后面没有对应的配置文件，所以可以自己创建这个配置文件：

```
cd /etc/rabbitmq

vim rabbitmq.config
```

在配置文件中写入如下内容：

```
[{rabbit, [{loopback_users, []}]}].
```

这里的意思是开放使用，rabbitmq默认创建的用户guest，密码也是guest，这个用户默认只能是本机访问，localhost或者127.0.0.1，从外部访问需要添加上面的配置。

保存配置后重启服务：

```
service rabbitmq-server restart
```

4、安装插件

查看已安装插件：

```
rabbitmq-plugins list
```

安装插件：

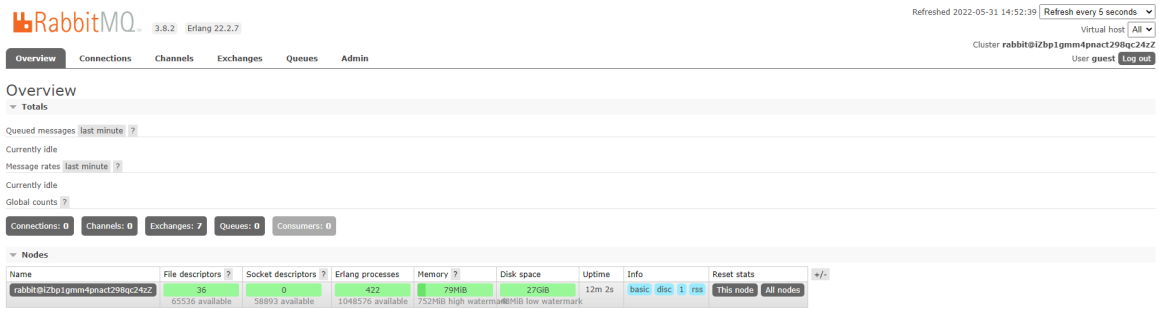
```
rabbitmq-plugins enable rabbitmq_management
```

重启rabbitmq服务：

> service rabbitmq-server restart

注意：需要在阿里云服务器配置安全策略中开放15672端口才可以使用web页面

通过http://ip:15672并使用guest,guest访问。



RabbitMQ角色

1.RabbitMQ的用户角色分类：
none、management、policymaker、monitoring、administrator
2.RabbitMQ各类角色描述：
none
不能访问 management plugin

management
用户可以通过AMQP做的任何事外加：
列出自己可以通过AMQP登入的virtual hosts
查看自己的virtual hosts中的queues, exchanges 和 bindings
查看和关闭自己的channels 和 connections
查看有关自己的virtual hosts的“全局”的统计信息，包含其他用户在这些virtual hosts中的活动。

policymaker
management可以做的任何事外加：
查看、创建和删除自己的virtual hosts所属的policies和parameters

monitoring
management可以做的任何事外加：
列出所有virtual hosts，包括他们不能登录的virtual hosts
查看其他用户的connections和channels
查看节点级别的数据如clustering和memory使用情况
查看真正的关于所有virtual hosts的全局的统计信息

administrator
policymaker和monitoring可以做的任何事外加:
创建和删除virtual hosts
查看、创建和删除users
查看创建和删除permissions
关闭其他用户的connections

3.创建用户并设置角色：
可以创建管理员用户，负责整个MQ的运维，例如：

rabbitmqctl add_user user_admin passwd_admin

赋予其administrator角色：

rabbitmqctl set_user_tags user_admin administrator

可以创建RabbitMQ监控用户，负责整个MQ的监控，例如：
#rabbitmqctl add_user user_monitoring passwd_monitor
赋予其monitoring角色：
#rabbitmqctl set_user_tags user_monitoring monitoring

可以创建某个项目的专用用户，只能访问项目自己的virtual hosts

rabbitmqctl add_user user_proj passwd_proj

赋予其monitoring角色：
#rabbitmqctl set_user_tags user_proj management

创建和赋角色完成后查看并确认：

rabbitmqctl list_users

4.RabbitMQ权限控制
默认virtual host："/"
默认用户：guest
guest具有"/"上的全部权限，仅能有localhost访问RabbitMQ包括Plugin，建议删除或更改密码。可通过将配置文件中loopback_users置孔来取消其本地访问的限制：
[{rabbit, [{loopback_users, []}]}]

用户仅能对其所能访问的virtual hosts中的资源进行操作。这里的资源指的是virtual hosts中的exchanges、queues等，操作包括对资源进行配置、写、读。配置权限可创建、删除、资源并修改资源的行为，写权限可向资源发送消息，读权限从资源获取消息。比如：
exchange和queue的declare与delete分别需要exchange和queue上的配置权限
exchange的bind与unbind需要exchange的读写权限
queue的bind与unbind需要queue写权限exchange的读权限
发消息(publish)需exchange的写权限
获取或清除(get、consume、purge)消息需queue的读权限
对何种资源具有配置、写、读的权限通过正则表达式来匹配，具体命令如下：
set_permissions [-p]
其中， 的位置分别用正则表达式来匹配特定的资源，如'^(amq.gen.*|amq.default)$'可以匹配server生成的和默认的exchange，'^$'不匹配任何资源

需要注意的是RabbitMQ会缓存每个connection或channel的权限验证结果、因此权限发生变化后需要重连才能生效。

为用户赋权：
$sudo rabbitmqctl set_permissions -p /vhost1 user_admin '.' '.' '.*'
该命令使用户user_admin具有/vhost1这个virtual host中所有资源的配置、写、读权限以便管理其中的资源
查看权限:
#rabbitmqctl list_user_permissions user_admin
Listing permissions for user "user_admin" ...

#rabbitmqctl list_permissions -p /vhost1
Listing permissions in vhost "/vhost1" ...

当连接超时时可能是防火墙禁止访问了，所以要开放该端口：

查询以开放端口：

> firewall-cmd --list-ports

开放 某个指定端口：

> firewall-cmd --zone=public --add-port=5672/tcp --permanent

重新加载：

> firewall-cmd --reload

即可。

或者直接关闭防火墙：

> systemctl stop firewalld（只执行这个，重启后不行，还必须执行systemclt disable firewalld）

RabbitMQ五种模式

1、简单模式

当生产者发送消息到交换机,交换机根据消息属性发送到队列，消费者监听绑定队列实现消息的接收和消费逻辑编写.简单模式下,强调的一个队列queue只被一个消费者监听消费.

应用场景：

手机短信,邮件单发

实现：

```
    //消费者
    @SneakyThrows
    @GetMapping("/hello-world")
    @ResponseBody
    public String producerHello() {
        //1、创建连接工厂
        ConnectionFactory connectionFactory = new ConnectionFactory();

        //2、设置参数，也可以在配置文件中设置
        connectionFactory.setHost("120.27.129.196"); //默认为localhost
        connectionFactory.setPort(5672);
        connectionFactory.setVirtualHost("/test"); //设置虚拟机，默认为"/“
        connectionFactory.setUsername("youxin"); //默认为guest
        connectionFactory.setPassword("1234"); //默认为guest

        //3、创建连接connection
        Connection connection = connectionFactory.newConnection();
        //4、创建频道 channel;
        Channel channel = connection.createChannel();
        //5、创建队列queue
        /*
        * queueDeclare(String queue, boolean durable, boolean exclusive, boolean
autoDelete,
                                    Map<String, Object> arguments)
        *  1、queue:队列名称
        *  2、durable:是否持久化，如果为true，当mq重启后消息还在
        *  3、exclusive:
        *       * 是否独占，只能有一个消费者监听这个队列
        *       * 当connection关闭时，是否删除队列
        *  4、autoDelete: 没有消费者使用时是否自动删除队列
        *  5、arguments: 其他属性（构造参数）
        *
        * */
        //如果没有"hello-world"这个队列则会自动创建该队列
```

```
        channel.queueDeclare("hello-world", true, false, false, null);
        //6、发送消息

        /*
        basicPublish(String exchange, String routingKey, BasicProperties props,
byte[] body)
        参数:
            1、exchange : 交换机名称，简单模式下会使用默认的""
            2、routingKey : 路由名称
            3、props : 其他配置信息，路由头等
            4、body : 消息体（队列中只能够传输字节类型）
         */
        channel.basicPublish("", "hello-world", null, "hello
rabbitmq!".getBytes());

        //7、关闭连接
        channel.close();
        connection.close();
        return "send rabbitmq queue!";
    }

    //生产者
    @ResponseBody
    @GetMapping("/received")
    @SneakyThrows
    public String receivedMessageQueue() {
        //1、创建连接工厂
        ConnectionFactory connectionFactory = new ConnectionFactory();

        //2、设置参数，也可以在配置文件中设置
            connectionFactory.setHost("120.27.129.196"); //默认为localhost
            connectionFactory.setPort(5672);
            connectionFactory.setVirtualHost("/test"); //设置虚拟机，默认为"/“
            connectionFactory.setUsername("youxin"); //默认为guest
            connectionFactory.setPassword("1234"); //默认为guest

        //3、创建连接connection
        Connection connection = connectionFactory.newConnection();
        //4、创建频道 channel;
        Channel channel = connection.createChannel();
        //5、创建队列queue
        /*
        * queueDeclare(String queue, boolean durable, boolean exclusive, boolean
autoDelete,
                                    Map<String, Object> arguments)
        *  1、queue:队列名称
        *  2、durable:是否持久化，如果为true，当mq重启后消息还在
        *  3、exclusive:
        *        * 是否独占，只能有一个消费者监听这个队列
        *        * 当connection关闭时，是否删除队列
        *  4、autoDelete: 没有消费者使用时是否自动删除队列
        *  5、arguments: 其他属性（构造参数）
        *
        * */
        //如果没有"hello-world"这个队列则会自动创建该队列
        channel.queueDeclare("hello-world", true, false, false, null);

        //6、接收消息
```

```
        /*
        basicConsume(String queue, boolean autoAck, Consumer callback)
        参数：
        queue: 队列名称
        autoAck : 自动接收消息
        callback : 回调函数
         */

        Consumer consumer = new DefaultConsumer(channel) {
            //就相当于一个对调方法
            /*
            handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body)
            参数：
            consumerTag : 消费者标识
            envelope : 获取信息，交换机，路由key...
            properties : 配置信息
            body : 消息体

             */
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
                System.out.println("consumerTag==>" + consumerTag);
                System.out.println("getExchange==>" + envelope.getExchange());
                System.out.println("getRoutingKey==>" +
envelope.getRoutingKey());
                System.out.println("getDeliveryTag==>" +
envelope.getDeliveryTag());
                System.out.println("properties==>" + properties.toString());
                System.out.println("body==>" + new String(body));

 System.out.println("==============================================");
            }
        };
        channel.basicConsume("hello-world", true, consumer);
        //因为是监听模式，所以可以不用关闭channel和connection
        //channel.close();
        //connection.close();
        return "received message!";
    }
```

2、工作模式

一个生产者，多个消费者，每个消费者获取到的消息唯一（消费者彼此竞争成为接收者）

生产者：发送消息到交换机
交换机：根据消息属性将消息发送给队列
消费者：多个消费者,同时绑定监听一个队列,之间形成了争抢消息的效果

应用场景：

抢红包、资源分配系统

基本实现：

```
    //生产者
    //work模式
    @SneakyThrows
    @ResponseBody
    @GetMapping("/work-producer")
    public String workProducer() {
        Connection connection = ConnectionUtils.getConnection();
        Channel channel = connection.createChannel();
        //如果没有"hello-world"这个队列则会自动创建该队列
        channel.queueDeclare("work", true, false, false, null);
        for (int i = 0; i < 100; i++) {
            channel.basicPublish("", "work", null, ("hello rabbitmq!--->" +
i).getBytes());
        }
        //7、关闭连接
        channel.close();
        connection.close();
        return "send rabbitmq queue!";
    }

    //消费者
    //work模式
    @SneakyThrows
    @ResponseBody
    @GetMapping("/work-consumer1")
    public String workConsumer1() {

        //创建频道
        Channel channel = connection.createChannel();
        //如果没有"hello-world"这个队列则会自动创建该队列
        channel.queueDeclare("work", true, false, false, null);
        Consumer consumer = new DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
                System.out.println("consumer1-received-body==>" + new
String(body) + "channelNumber==>" + channel.getChannelNumber()) ;
                //进行手动应答
                /*
                 * 第一个参数：自动应答
                 * 第二个参数：false表示收到消息了
                 */
                channel.basicAck(envelope.getDeliveryTag(),false);
            }
        };
        channel.basicConsume("work", false, consumer);
        return "consumer1 received!";
    }

    //work模式
    @SneakyThrows
    @ResponseBody
    @GetMapping("/work-consumer2")
    public String workConsumer2() {
        //创建频道
        Channel channel = connection.createChannel();
        //如果没有"hello-world"这个队列则会自动创建该队列
```

```
        channel.queueDeclare("work", true, false, false, null);
        Consumer consumer = new DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
                System.out.println("consumer2-received-body==>" + new
String(body) + "channelNumber==>" + channel.getChannelNumber());
                //进行手动应答
                /*
                 * 第一个参数：自动应答
                 * 第二个参数：false表示收到消息了
                 */
                channel.basicAck(envelope.getDeliveryTag(),false);
            }
        };
        channel.basicConsume("work", false, consumer);
        return "consumer2 received!";
    }
```

实现结果：

```
consumer1-received-body==>hello rabbitmq!--->0channelNumber==>3
consumer2-received-body==>hello rabbitmq!--->1channelNumber==>4
consumer1-received-body==>hello rabbitmq!--->2channelNumber==>3
consumer2-received-body==>hello rabbitmq!--->3channelNumber==>4
consumer1-received-body==>hello rabbitmq!--->4channelNumber==>3
consumer2-received-body==>hello rabbitmq!--->5channelNumber==>4
consumer1-received-body==>hello rabbitmq!--->6channelNumber==>3
consumer2-received-body==>hello rabbitmq!--->7channelNumber==>4
```

3、发布\订阅者模式

一个生产者发送的消息会被多个消费者获取。

交换机为fanout

生产端：发送消息到交换机
交换机：由于是发布订阅模式,会将这个消息发送同步到后端所有与其绑定的队列
消息端：简单模式 1个队列绑定一个消费者 争抢模式 1个队列绑定多个消费者

应用场景
邮件的群发,广告的群发

基本实现：

```
    // 发布/订阅者模式
    @SneakyThrows
    @ResponseBody
    @GetMapping("/publish-producer")
    public String publishProducer() {
        Connection connection = ConnectionUtils.getConnection();
        Channel channel = connection.createChannel();
        //申明交换机
        /*
         * 第一个参数：交换机的名字
```

```
         * 第二个参数：交换机的类型
         *       如果使用的是发布订阅模式：只能写fanout
         */
        channel.exchangeDeclare("publish-exchange","fanout");

        //发送消息到交换机
        for (int i = 0; i < 100; i++) {
            channel.basicPublish("publish-exchange", "", null, ("hello
rabbitmq!--->" + i).getBytes());
        }
        channel.close();
        connection.close();
        return "send rabbitmq exchange!";
    }

    //消费者
    //发布订阅模式：生产者没有将消息发送到队列而是发送到交换机，每个消费者都有自己的队列，每个队
列都要绑定到交换机，消费者获取到生产者发送的信息是完整的
    @SneakyThrows
    @ResponseBody
    @GetMapping("/publish-consumer1")
    public String publishConsumer1() {
        Channel channel = connection.createChannel();
        //申明队列
        channel.queueDeclare("publish1", true, false, false, null);
        //申明交换机
        channel.exchangeDeclare("publish-exchange", "fanout");
        /*将队列绑定到交换机
         * 第一个参数：队列的名字
         * 第二个参数：交换机的名字
         * 第三个参数：路由的key
         */
        channel.queueBind("publish1", "publish-exchange", "");
        Consumer consumer = new DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
                System.out.println("consumer1--receivedMessage===>" + new
String(body));
            }
        };
        channel.basicConsume("publish1", true, consumer);
        return "publish consumer1 received!";
    }

    @SneakyThrows
    @ResponseBody
    @GetMapping("/publish-consumer2")
    public String publishConsumer2() {
        Channel channel = connection.createChannel();
        //申明队列
        channel.queueDeclare("publish2", true, false, false, null);
        //申明交换机
        channel.exchangeDeclare("publish-exchange", "fanout");
        /*将队列绑定到交换机
         * 第一个参数：队列的名字
         * 第二个参数：交换机的名字
         * 第三个参数：路由的key
```

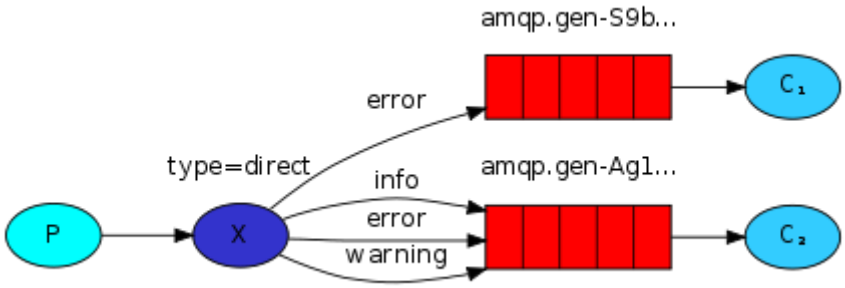
```
         */
        channel.queueBind("publish2", "publish-exchange", "");
        Consumer consumer = new DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
                System.out.println("consumer2--receivedMessage===>" + new
String(body));
            }
        };
        channel.basicConsume("publish2", true, consumer);
        return "publish consumer2 received!";
    }
```

实现结果：

```
consumer2--receivedMessage===>hello rabbitmq!--->0
consumer2--receivedMessage===>hello rabbitmq!--->1
consumer1--receivedMessage===>hello rabbitmq!--->0
consumer1--receivedMessage===>hello rabbitmq!--->1
consumer1--receivedMessage===>hello rabbitmq!--->2
consumer2--receivedMessage===>hello rabbitmq!--->2
consumer2--receivedMessage===>hello rabbitmq!--->3
consumer2--receivedMessage===>hello rabbitmq!--->4
consumer1--receivedMessage===>hello rabbitmq!--->3
```

4、路由模式

将消息携带routing key和队列绑定的routing key，如果匹配上了，就把消息发送给队列。



基本实现：

```
    //生产者
    //路由模式
    @SneakyThrows
    @ResponseBody
    @GetMapping("route-producer")
    public String routeProducer() {
        Connection connection = ConnectionUtils.getConnection();
        Channel channel = connection.createChannel();
        //如果是路由模式，第二个参数只能为direct
        channel.exchangeDeclare("route-exchange", "direct");
```

```
        //发送消息
        for (int i = 0; i < 100; i++) {
            if (i % 2 == 0) {
                //如果是偶数,将消息发送到路由键为route1队列中
                channel.basicPublish("route-exchange", "route1", null, ("hello
rabbitmq!--->" + i).getBytes());
            }else {
                //如果是奇数，将消息发送到路由键为route2队列中
                channel.basicPublish("route-exchange", "route2", null, ("hello
rabbitmq!--->" + i).getBytes());
            }
        }
        channel.close();
        connection.close();
        return "send rabbitmq route!";
    }

    //消费者
    //路由模式:
    @SneakyThrows
    @ResponseBody
    @GetMapping("/route-consumer1")
    public String routeConsumer1() {
        Channel channel = connection.createChannel();
        //队列申明
        channel.queueDeclare("route-queue1", true, false, false, null);
        //交换机申明
        channel.exchangeDeclare("route-exchange", "direct");

        //绑定队列到交换机,接收偶数队列中的消息
        channel.queueBind("route-queue1", "route-exchange", "route1");
        Consumer consumer = new DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
                System.out.println("consumer1--receivedMessage===>" + new
String(body));
            }
        };
        channel.basicConsume("route-queue1", true, consumer);
        return "route consumer1 received!";
    }

    @SneakyThrows
    @ResponseBody
    @GetMapping("/route-consumer2")
    public String routeConsumer2() {
        Channel channel = connection.createChannel();
        //队列申明
        channel.queueDeclare("route-queue2", true, false, false, null);
        //交换机申明
        channel.exchangeDeclare("route-exchange", "direct");

        //绑定队列到交换机,接收偶数队列中的消息
        channel.queueBind("route-queue2", "route-exchange", "route2");
        Consumer consumer = new DefaultConsumer(channel) {
            @Override
```

```
            public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
                System.out.println("consumer2--receivedMessage===>" + new
String(body));
            }
        };
        channel.basicConsume("route-queue2", true, consumer);
        return "route consumer2 received!";
    }
```

实现结果：

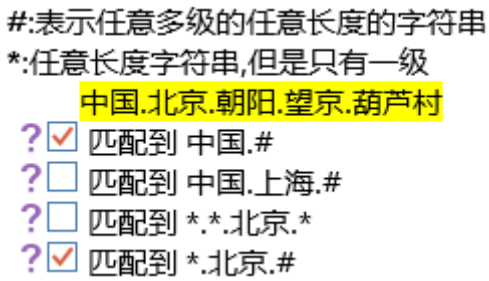
```
consumer1--receivedMessage===>hello rabbitmq!--->0
consumer1--receivedMessage===>hello rabbitmq!--->2
consumer2--receivedMessage===>hello rabbitmq!--->1
consumer1--receivedMessage===>hello rabbitmq!--->4
consumer2--receivedMessage===>hello rabbitmq!--->3
consumer1--receivedMessage===>hello rabbitmq!--->6
consumer2--receivedMessage===>hello rabbitmq!--->5
consumer2--receivedMessage===>hello rabbitmq!--->7
```

5、topic模式

非常类似于路由模式的结构,区别在于后端队列绑定交换机使用的路由key.

topic模式相当于对路由模式的升级，topic模式主要就是在匹配的规则上可以实现模糊匹配

`*`:只能匹配一个单词　　如：user.*　user.username
`#`：可以匹配一个到多个单词 如：user.#　user.username.xbb



应用场景
实现多级传递的路由筛选工作.

基本实现：

```
    //生产者
    //topic模式
    @SneakyThrows
    @ResponseBody
    @GetMapping("topic-producer")
    public String topicProducer() {
        Connection connection = ConnectionUtils.getConnection();
        Channel channel = connection.createChannel();
//        channel.queueDeclare("topic-queue", true, false, false, null);
        //申明交换机
```

```
        channel.exchangeDeclare("topic-exchange", "topic");
        for (int i = 0; i < 100; i++) {
            channel.basicPublish("topic-exchange", "hello.world.world", null,
("topic模式" + i).getBytes());
        }
        channel.close();
        connection.close();
        return "send rabbitmq topic!";
    }

    //消费者
    @SneakyThrows
    @ResponseBody
    @GetMapping("/topic-consumer1")
    public String topicConsumer1() {
        Channel channel = connection.createChannel();
        //申明队列
        channel.queueDeclare("topic-queue", true, false, false, null);
        //申明交换机
        channel.exchangeDeclare("topic-exchange", "topic");
        //绑定
        channel.queueBind("topic-queue", "topic-exchange", "hello.#");
        Consumer consumer = new DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
                System.out.println("topicConsumer1--hello--ReceivedMessage==>" +
new String(body));
            }
        };

        channel.basicConsume("topic-queue", true, consumer);
        return "topic consumer1 received!";
    }

    @SneakyThrows
    @ResponseBody
    @GetMapping("/topic-consumer2")
    public String topicConsumer2() {
        Channel channel = connection.createChannel();
        //申明队列
        channel.queueDeclare("topic-queue", true, false, false, null);
        //申明交换机
        channel.exchangeDeclare("topic-exchange", "topic");
        //绑定
        channel.queueBind("topic-queue", "topic-exchange", "*.world");
        Consumer consumer = new DefaultConsumer(channel) {

            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
                System.out.println("topicConsumer2--world--ReceivedMessage==>" +
new String(body));
            }
        };

        channel.basicConsume("topic-queue", true, consumer);
        return "topic consumer2 received!";
```

```
    }
```

实现结果：

```
topicConsumer1--hello--ReceivedMessage==>topic模式0
topicConsumer1--hello--ReceivedMessage==>topic模式2
topicConsumer1--hello--ReceivedMessage==>topic模式4
topicConsumer2--world--ReceivedMessage==>topic模式1
topicConsumer2--world--ReceivedMessage==>topic模式3
topicConsumer2--world--ReceivedMessage==>topic模式5
topicConsumer2--world--ReceivedMessage==>topic模式7
topicConsumer2--world--ReceivedMessage==>topic模式9
```

三种交换机：

direct（路由） —>对应路由模式 —>100%routing key匹配上了，交换机才会把数据发送给队列
fanout（发布订阅） —>对应发布订阅模式 —>只要队列发布了交换机，就会把数据发送给队列，不会去管routing key是
否匹配
topic（主题）—>对应主题模式 —>可以定义多级路径，可以进行模糊匹配。

RabbitMQ中的高级属性

1、confirm机制

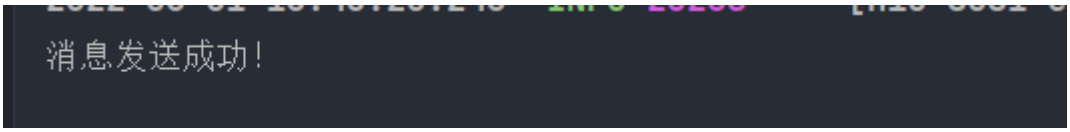
放到队列中的消息，怎么保证一定成功的放入队列了呢?
confirm机制：只要消息放入队列成功，那么队列就一定会给反馈

基本实现：

```
    //生产者
    //confirm机制
    @SneakyThrows
    @ResponseBody
    @GetMapping("/confirm-producer")
    public String confirmProducer() {
        Connection connection = ConnectionUtils.getConnection();
        Channel channel = connection.createChannel();
        //开启confirm消息机制
        channel.confirmSelect();
        //对消息进行监听
        channel.addConfirmListener(new ConfirmListener() {
            @Override
            public void handleAck(long deliveryTag, boolean multiple) throws
IOException {
                System.out.println("消息发送成功！");
            }

            @Override
            public void handleNack(long deliveryTag, boolean multiple) throws
IOException {
```

```
                System.out.println("消息发送失败！");
            }
        });

        channel.queueDeclare("hello-world", true, false, false, null);
        channel.basicPublish("", "hello-world", null, "hello,
world!".getBytes());
        //想要监听到发送的消息，不能够将channel和connection关闭
//        channel.close();
//        connection.close();
        return "send confirm rabbitmq message!";
    }
```

结果：



2、return机制

生产者发送消息的时候，如果交换机不存在，或者路由的key不存在，这时候就需要监听这种到达不了的消息。
注意：当前的队列中必须要有消费者存在

基本实现：

```
    //生产者
    //return机制
    @SneakyThrows
    @ResponseBody
    @GetMapping("return-producer")
    public String returnProducer() {
        Connection connection = ConnectionUtils.getConnection();
        Channel channel = connection.createChannel();
        channel.addReturnListener(new ReturnListener() {

            /***
             * @author youxin
             * @date 2022-06-01 16:59
             * @param replyCode 队列相应给浏览器的状态码
             * @param replyText 状态码对于的文本信息
             * @param exchange 交换机的名字
             * @param routingKey 路由的key
             * @param properties 消息的属性
             * @param body 消息体的内容
             * @return void
             * @throws IOException
             * @since
             */
            @Override
            public void handleReturn(int replyCode, String replyText, String
exchange, String routingKey, AMQP.BasicProperties properties, byte[] body)
throws IOException {
                System.out.println("监听到不可达的消息");
                System.out.println("状态码" + replyCode);
                System.out.println("文本信息" + replyText);
```

```
                System.out.println("交换机名字" + exchange);
                System.out.println("body内容" + new String(body));
            }
        });
        //这里的第三个参数如果设置为true，表示要监听不可达的消息进行处理
        //如果设置为false，那么队列会直接删除这个消息
        channel.basicPublish("topic-exchange", "route-key", true, null, "return机
制".getBytes());
        return "return rabbitmq message!";
    }
```

结果：

3、消费端的限流问题

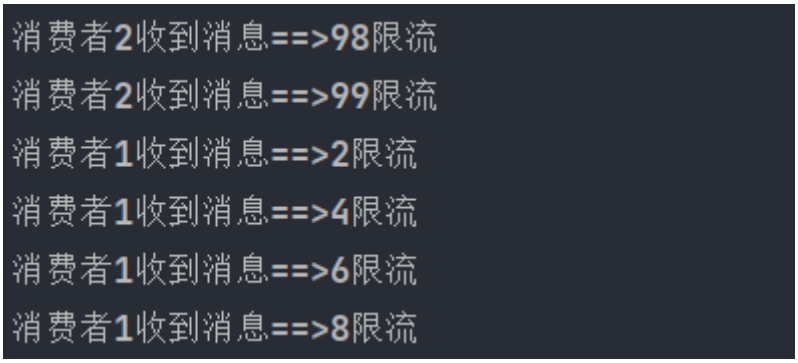
假设消费者挂了，消息全部堆积到队列里面，然后当消费者重新启动时，队列里的消息就全部发送过来，但是客户端没办法同时去处理那么多的消息
这种场景下就需要对消费者进行限流。

基本实现：

```
    //生产者
    //高级属性-限流问题
    @SneakyThrows
    @ResponseBody
    @GetMapping("limit-producer")
    public String limitProducer() {
        Connection connection = ConnectionUtils.getConnection();
        Channel channel = connection.createChannel();
        channel.queueDeclare("limit-test", true, false, false, null);
        for (int i = 0; i < 100; i++) {
            channel.basicPublish("", "limit-test", null, (i + "限
流").getBytes());
        }
        channel.close();
        connection.close();
        return "send limit rabbitmq message!";
    }

    //消费者
    //限流消费者
    @SneakyThrows
    @ResponseBody
    @GetMapping("/limit-consumer1")
    public String limitConsumer1() {
        Connection connection = ConnectionUtils.getConnection();
        Channel channel = connection.createChannel();
        channel.queueDeclare("limit-test", true, false, false, null);
```

```
        /*
         * 开启限流
         * 第一个参数：消息本身的大小，如果设置为0，那么表示对消息大小不限制
         * 第二个参数：一次性推送消息的最大数量，前提消息必须手动应答完成
         * 第三个参数：true：将设置应用到通道  false：只是当前消费者的策略
         */
        channel.basicQos(0, 5, false);
        Consumer consumer = new DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
                System.out.println("消费者1收到消息==>" + new String(body));
                try {
                    Thread.sleep(1000);
                }catch (Exception e) {
                    e.printStackTrace();
                }
                //进行手动应答
                channel.basicAck(envelope.getDeliveryTag(), false);
            }
        };
        //使用限流必须手动应答
        channel.basicConsume("limit-test", false, consumer);
        return "consumer1 received message!";
    }

    @SneakyThrows
    @ResponseBody
    @GetMapping("/limit-consumer2")
    public String limitConsumer2() {
        Connection connection = ConnectionUtils.getConnection();
        Channel channel = connection.createChannel();
        channel.queueDeclare("limit-test", true, false, false, null);
        Consumer consumer = new DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
                System.out.println("消费者2收到消息==>" + new String(body));
                //进行手动应答
                channel.basicAck(envelope.getDeliveryTag(), false);
            }
        };
        //使用限流必须手动应答
        channel.basicConsume("limit-test", false, consumer);
        return "consumer2 received message!";
    }
```

实现结果：消费者1每隔一秒接收一条消息，并且总共只接收5条消息，剩下的消息全部交由消费者2接收。



4、ttl队列

给队列中的消息添加时间限制，如果超过时间限制这个消息就会被删除。

基本实现：

```
    //生产者
    //ttl消息队列：给队列中的消息添加时间限制，如果超过时间限制这个消息就会被删除。
    @SneakyThrows
    @ResponseBody
    @GetMapping("/ttl-producer")
    public String ttlProducer() {
        Connection connection = ConnectionUtils.getConnection();
        Channel channel = connection.createChannel();
        Map<String, Object> argus = new HashMap<>();
        //设置5秒过期时间
        argus.put("x-message-ttl", 5000);
        //这里因为设置了过期时间，所以不能够将持久化置为true
        channel.queueDeclare("ttl-queue", false, false, false, argus);
        for (int i = 0; i < 100; i++) {
            channel.basicPublish("", "ttl-queue", null, ("ttl-queue==>" +
i).getBytes());
        }
        channel.close();
        connection.close();
        return "send ttl-message!";
    }
```

注意：在生产者端设置了ttl过期时间，则在消费者端同样需要设置ttl过期时间

结果：

在生产者发出消息的5秒后再通过消费者监听，消息过期，消费者无法获取消息。

5、死信队列

什么是死信队列

当消息在队列中变成死信之后、可以定义它重新push 到另外一个交换机上、这个交换机 也有自己对应的队列 这个队列就称为死信队列
满足死信的条件：

发送到队列中的消息被拒绝了
消息的ttl时间过期
队列达到了最大长度 再往里面放信息
当这个队列中如果有这个死信的时候、rabbitmq就会将这个消息自动发送到我们提前定义好的死信队列

中去
基本实现：

```
    //生产者
    //死信队列
    @SneakyThrows
    @ResponseBody
    @GetMapping("/dlx-producer")
    public String dlcProducer() {
        Connection connection = ConnectionUtils.getConnection();
        Channel channel = connection.createChannel();
        channel.basicPublish("ttl-dlx-exchange", "dlx.hello", false,null, "死信队
列消息".getBytes());
        return "send dlx message!";
    }

    //消费者
    //死信队列消费者
    @SneakyThrows
    @ResponseBody
    @GetMapping("/dlx-consumer")
    public String dlxConsumer() {
        Connection connection = ConnectionUtils.getConnection();
        Channel channel = connection.createChannel();
        //绑定交换机
        channel.exchangeDeclare("ttl-dlx-exchange", "topic");
        //申明队列
        Map<String, Object> argus = new HashMap<>();
        argus.put("x-message-ttl", 5000);
        argus.put("x-dead-letter-exchange", "dlx-exchange");
        channel.queueDeclare("ttl-dlx-queue", false, false, false, argus);
        channel.queueBind("ttl-dlx-queue", "ttl-dlx-exchange", "dlx.*");

        //绑定死信队列
        channel.exchangeDeclare("dlx-exchange", "topic");
        //死信队列里面的arguments不设置属性
        channel.queueDeclare("dlx-queue", false, false, false, null);
        channel.queueBind("dlx-queue", "dlx-exchange", "dlx.*");

        Consumer consumer = new DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
                System.out.println("消费者收到消息==>" + new String(body));
                System.out.println("获取消息的交换机名==>" +
envelope.getExchange());
            }
        };

        channel.basicConsume("dlx-queue", true, consumer);
        return "dlx consumer received message!";
    }
```

结果：

```
消费者收到消息==>死信队列消息
获取消息的交换机名==>dlx-exchange
```

6、延迟队列

延迟队列：即消息进入队列后不会被立即消费，而是到达指定时间后，才会被消费。

需求：下单后30分钟未支付，取消订单，回滚库存。

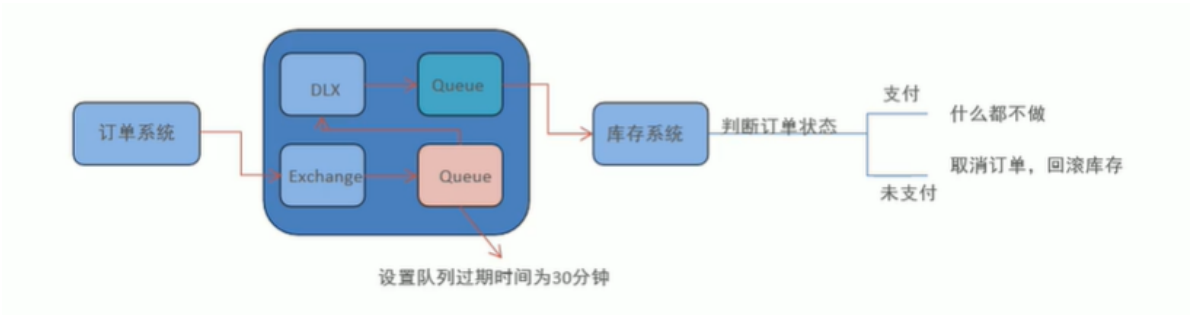
实现方式：

1、定时任务

2、延迟队列

在RabbitMQ中并没有提供延迟队列功能。

所以使用ttl+死信队列组合实现延迟队列的效果。



具体实现就是上面的死信队列的实现过程。

7、消费者端的手动签收和重回消息队列

消费者除了手动签收应答还可以拒绝接受消息，让消息重回队列，即消费者可以收到消息，但是拒绝签收，所以设置requeue为true后将把消息重回消息队列中。

基本实现：

```
    //生产者
    //消息手动签收和消息重回队列
    @SneakyThrows
    @ResponseBody
    @GetMapping("/refuse-producer")
    public String refuseProducer() {
        Connection connection = ConnectionUtils.getConnection();
        Channel channel = connection.createChannel();
        channel.queueDeclare("refuse-queue", false, false, false, null);
        channel.basicPublish("", "refuse-queue", null, "拒绝消息".getBytes());
        channel.close();
        connection.close();
        return "send refuse message!";
    }
```

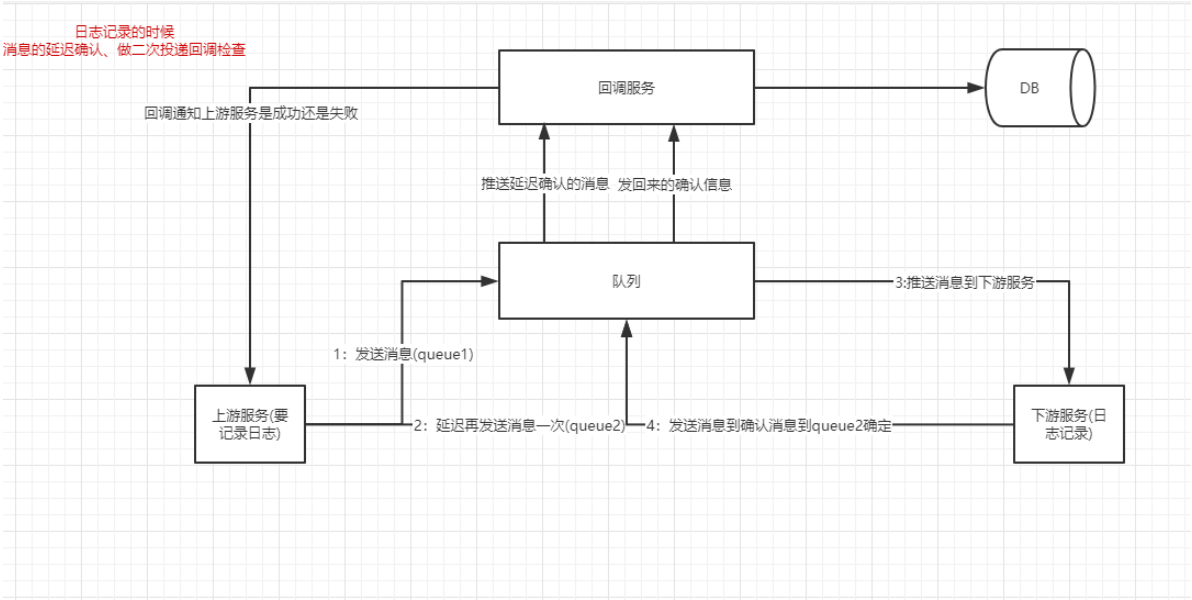
```
    //消费者
    //拒绝接收消息和消息重入队列
    @SneakyThrows
    @ResponseBody
    @GetMapping("/refuse-consumer")
    public String refuseConsumer() {
        Connection connection = ConnectionUtils.getConnection();
        Channel channel = connection.createChannel();
        channel.queueDeclare("refuse-queue", false, false, false, null);
        //消费者申明
        Consumer consumer = new DefaultConsumer(channel){
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
                System.out.println("消费者收到消息==>" + new String(body));
                //拒绝消息
                /*
                 * 第一个参数：自动应答
                 * 第二个参数：true拒绝所有消息，false则拒绝提供的消息
                 * 第三个参数：表示决绝签收之后这个消息是否要重回队列?
                 */
                channel.basicNack(envelope.getDeliveryTag(), false, true);
            }
        };
        //绑定消费者
        /*
         * 第一个参数：队列名字
         * 第二个参数：是否自动应答
         * 第三个参数：消费者
         */
        channel.basicConsume("refuse-queue", false, consumer);

        return "refuse consumer received message!";
    }
```

结果：消费者接收到消息后拒绝签收消息，又将消息返回到消息队列中，此时消息队列中的消息依旧存在并没有被消费。

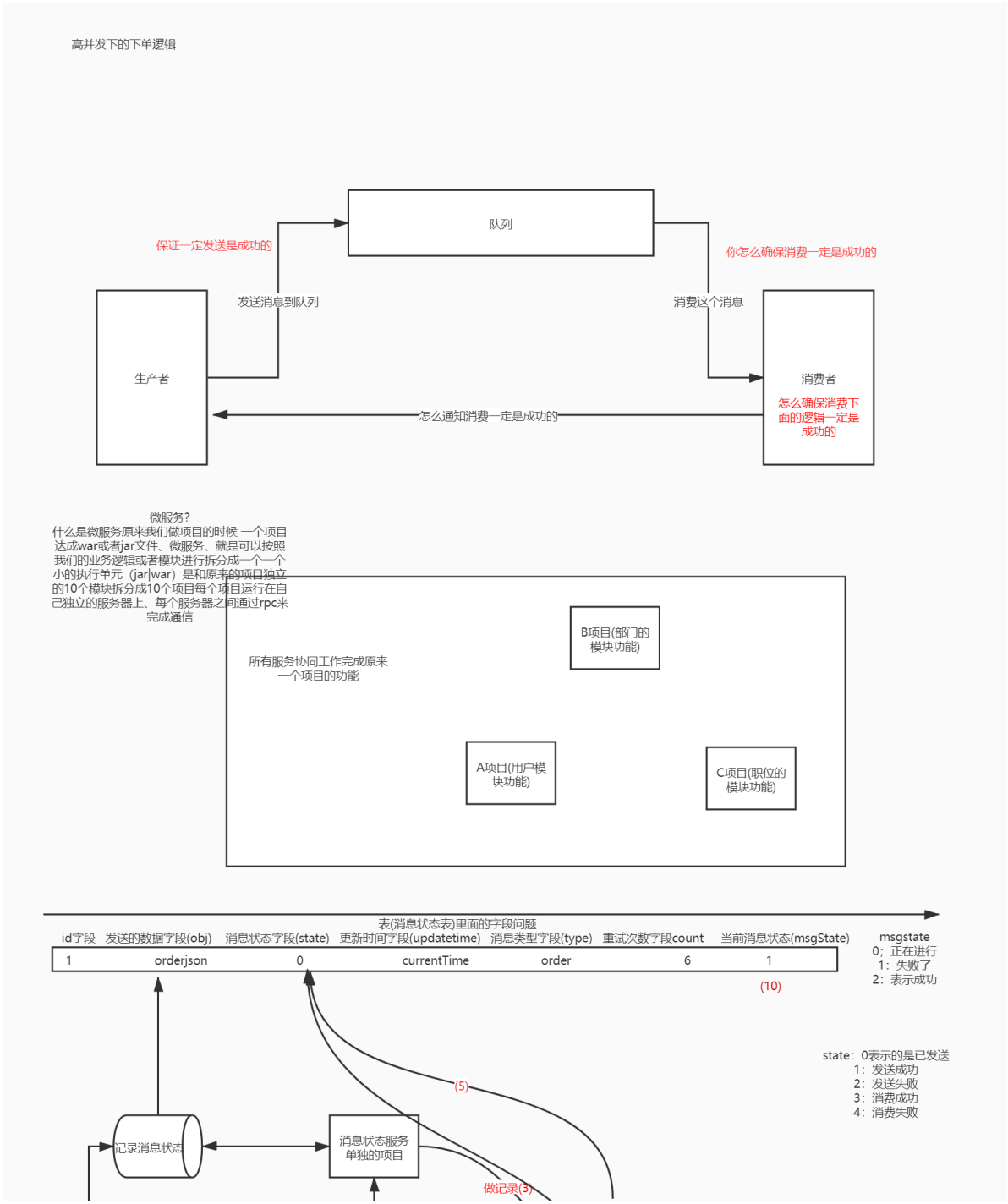
/test	refuse-queue	classic		0	idle	1	0	1	0.00/s	0.00/s	0.00/s

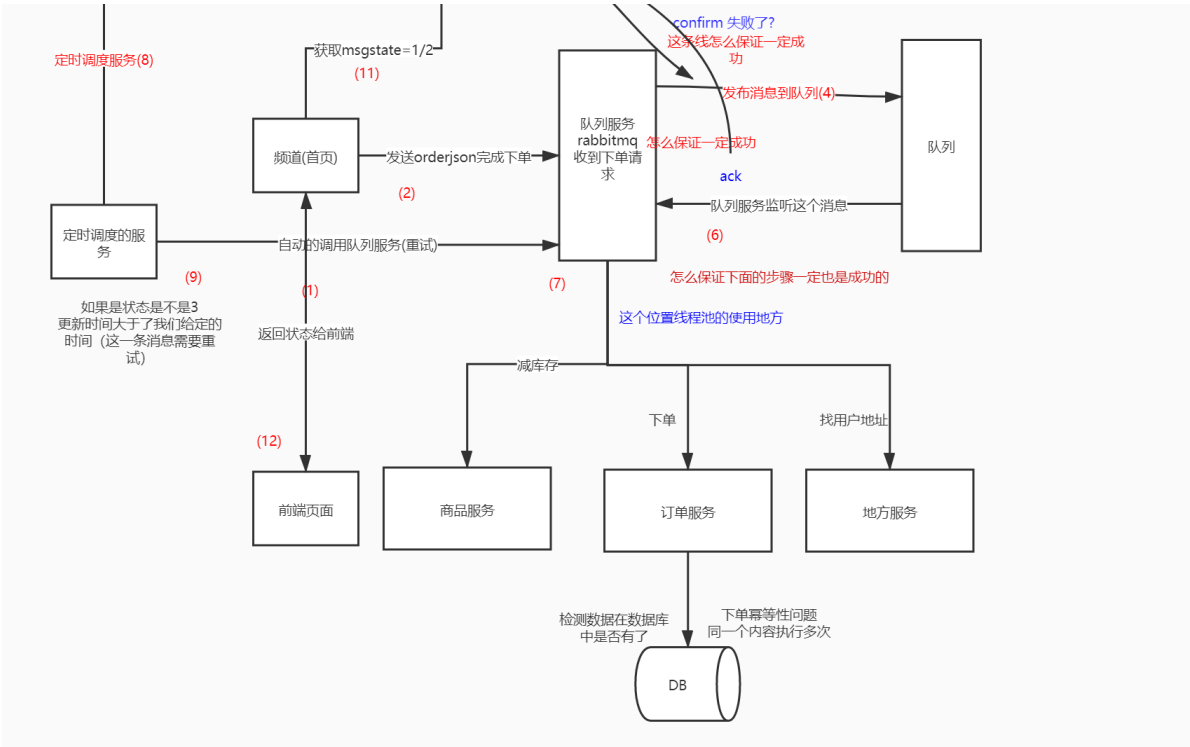
8、保证消息的投递是成功的

1.消息的延迟投递来解决传递的可靠性



2.日志消息表实现可靠消息的传输





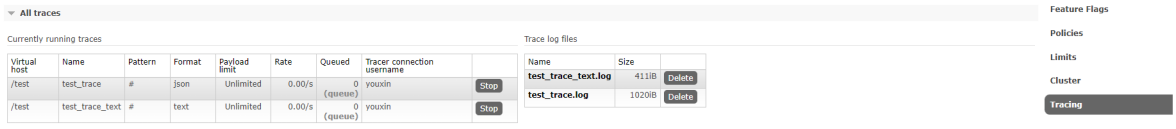
9、消息追踪

开启trace插件：

> rabbitmq-plugins list 查看所有插件

> rabbitmq-plugins enable rabbitmq_tracing 打开插件

这时在web管理界面的admin中可以看到trace选项，创建日志记录后，就可以对每条消息进行跟踪了



> rabbitmq-plugins disable rabbitmq_tracing 关闭插件