RabbitMQ 内容汇总

RabbitMQ 内容汇总
- 什么叫消息队列
- 为什么要用消息队列
- RabbitMQ 简介
- RabbitMQ 特点
- RabbitMQ 安装和使用
  - **一、安装依赖环境**
  - **二、安装RabbitMQ**
  - **三、启动和关闭**
  - **四、开启Web管理插件**
  - **五、防火墙添加端口**
- RabbitMQ基本配置
- RabbitMQ管理界面
  - RabbitMQ 角色分类
- hello world
- AMQP 协议
- RabbitMQ中的概念模型
  - 消息模型
  - RabbitMQ 基本概念
  - AMQP 中的消息路由
  - Exchange 类型
- RabbitMQ 集群部署方式
  - **多机多节点集群部署**
    - **一、 环境准备**
    - **二、修改配置文件**
    - **三、防火墙添加端口**
    - **四、启动RabbitMQ**
  - **单机多节点部署**
    - **一、环境准备**
    - **二、防火墙添加端口**
    - **三、启动RabbitMQ**
- **镜像队列模式集群**
- **示例说明**
  - **Virtual Hosts**
    - **Virtual Hosts 的功能说明**
    - **vhost使用示例**
  - **集群连接恢复**
  - **重连测试方式**
  - **镜像队列测试**
- RabbitMQ持久化机制、内存磁盘控制
  - 示例说明
  - 持久化
  - 内存告警
    - **模拟内存告警**
  - 内存换页
  - 磁盘告警
    - **模拟磁盘告警**
- 消息可靠性
  - 消息的可靠投递
    - 生产端
    - 消费端
  - 消费端限流

TTL
死信队列
延迟队列
RabbitMQ 监控
Management UI
rabbitmqctl 命令
REST API
prometheus + grafana 监控rabbitmq
前言
一、使用独立程序来获取指标（RabbitMQ_exporter）
安装rabbitmq_exporter
使用docker-compose方式启动prometheus和grafana
Prometheus配置
Grafana配置
二、RabbitMQ内部集成Prometheus来获取指标

什么叫消息队列

消息（Message）是指在应用间传送的数据。消息可以非常简单，比如只包含文本字符串，也可以更复杂，可能包含嵌入对象。

消息队列（Message Queue）是一种应用间的通信方式，消息发送后可以立即返回，由消息系统来确保消息的可靠传递。消息发布者只管把消息发布到 MQ 中而不用管谁来取，消息使用者只管从 MQ 中取消息而不管是谁发布的。这样发布者和使用者都不用知道对方的存在。

为什么要用消息队列

从上面的描述中可以看出消息队列是一种应用间的异步协作机制，那什么时候需要使用 MQ 呢?

以常见的订单系统为例，用户点击【下单】按钮之后的业务逻辑可能包括：扣减库存、生成相应单据、发红包、发短信通知。在业务发展初期这些逻辑可能放在一起同步执行，随着业务的发展订单量增长，需要提升系统服务的性能，这时可以将一些不需要立即生效的操作拆分出来异步执行，比如发放红包、发短信通知等。这种场景下就可以用 MQ ，在下单的主流程（比如扣减库存、生成相应单据）完成之后发送一条消息到 MQ 让主流程快速完结，而由另外的单独线程拉取MQ的消息（或者由 MQ 推送消息），当发现 MQ 中有发红包或发短信之类的消息时，执行相应的业务逻辑。

以上是用于业务解耦的情况，其它常见场景包括最终一致性、广播、错峰流控等等。

RabbitMQ 简介

RabbitMQ 是一个开源的AMQP实现，服务器端用Erlang语言编写，支持多种客户端。用于在分布式系统中存储转发消息，在易用性、扩展性、高可用性等方面表现不俗。

RabbitMQ 特点

AMQP ： Advanced Message Queue，高级消息队列协议。它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，并不受产品、开发语言等条件的限制。

RabbitMQ 最初起源于金融系统，用于在分布式系统中存储转发消息，在易用性、扩展性、高可用性等方面表现不俗。具体特点包括：

1. 可靠性（Reliability）
   RabbitMQ 使用一些机制来保证可靠性，如持久化、传输确认、发布确认。

2. 灵活的路由（Flexible Routing）
   在消息进入队列之前，通过 Exchange 来路由消息的。对于典型的路由功能，RabbitMQ 已经提供了一些内置的 Exchange 来实现。针对更复杂的路由功能，可以将多个 Exchange 绑定在一起，也通过插件机制实现自己的 Exchange 。
3. 消息集群（Clustering）
   多个 RabbitMQ 服务器可以组成一个集群，形成一个逻辑 Broker 。
4. 高可用（Highly Available Queues）
   队列可以在集群中的机器上进行镜像，使得在部分节点出问题的情况下队列仍然可用。
5. 多种协议（Multi-protocol）
   RabbitMQ 支持多种消息队列协议，比如 STOMP、MQTT 等等。
6. 多语言客户端（Many Clients）
   RabbitMQ 几乎支持所有常用语言，比如 Java、.NET、Ruby 等等。
7. 管理界面（Management UI）
   RabbitMQ 提供了一个易用的用户界面，使得用户可以监控和管理消息 Broker 的许多方面。
8. 跟踪机制（Tracing）
   如果消息异常，RabbitMQ 提供了消息跟踪机制，使用者可以找出发生了什么。
9. 插件机制（Plugin System）
   RabbitMQ 提供了许多插件，来从多方面进行扩展，也可以编写自己的插件。

RabbitMQ 安装和使用

一、安装依赖环境

1. 在 http://www.rabbitmq.com/which-erlang.html 页面查看安装rabbitmq需要安装erlang对应的版本
2. 在 https://github.com/rabbitmq/erlang-rpm/releases 页面找到需要下载的erlang版本，`erlang-*.centos.x86_64.rpm` 就是centos版本的。
3. 复制下载地址后，使用wget命令下载

```
wget -P /home/download https://github.com/rabbitmq/erlang-rpm/releases/download/v23.3.3/erlang-23.3.3-1.el7.x86_64.rpm
```

4. 安装 Erlang

```
sudo rpm -Uvh /home/download/erlang-21.2.3-1.el7.centos.x86_64.rpm
```

5. 安装 socat

```
sudo yum install -y socat
```

二、安装RabbitMQ

1. 在官方下载页面找到CentOS7版本的下载链接，下载rpm安装包

```
wget -P /home/download https://github.com/rabbitmq/rabbitmq-server/releases/download/v3.8.15/rabbitmq-server-3.8.15-1.el7.noarch.rpm
```

`提示：可以在 https://github.com/rabbitmq/rabbitmq-server/tags 下载历史版本`

2. 安装RabbitMQ

```
sudo rpm -Uvh /home/download/rabbitmq-server-3.7.9-1.el7.noarch.rpm
```

三、启动和关闭

- 启动服务

```
sudo systemctl start rabbitmq-server
```

- 查看状态

```
sudo systemctl status rabbitmq-server
```

- 停止服务

```
sudo systemctl stop rabbitmq-server
```

- 设置开机启动

```
sudo systemctl enable rabbitmq-server
```

四、开启Web管理插件

1. 开启插件

```
rabbitmq-plugins enable rabbitmq_management
```

`说明：rabbitmq有一个默认的guest用户，但只能通过localhost访问，所以需要添加一个能够远程访问的用户。`

2. 添加用户

```
rabbitmqctl add_user admin admin
```

3. 为用户分配操作权限

```
rabbitmqctl set_user_tags admin administrator
```

4. 为用户分配资源权限

```
rabbitmqctl set_permissions -p / admin ".*" ".*" ".*"
```

五、防火墙添加端口

- RabbitMQ 服务启动后，还不能进行外部通信，需要将端口添加都防火墙

1. 添加端口

```
sudo firewall-cmd --zone=public --add-port=4369/tcp --permanent
sudo firewall-cmd --zone=public --add-port=5672/tcp --permanent
sudo firewall-cmd --zone=public --add-port=25672/tcp --permanent
sudo firewall-cmd --zone=public --add-port=15672/tcp --permanent
```

2. 重启防火墙

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

RabbitMQ基本配置

RabbitMQ 有一套默认的配置，能够满足日常开发需求，如果需要修改，需要自己创建一个配置文件

touch /etc/rabbitmq/rabbitmq.conf

配置文件示例：
https://github.com/rabbitmq/rabbitmq-server/blob/master/deps/rabbit/docs/rabbitmq.conf.example

配置项说明：
https://www.rabbitmq.com/configure.html#config-items

RabbitMQ 会绑定一些端口，安装完后，需要将这些端口添加至防火墙。

- 4369：是Erlang的端口/结点名称映射程序，用来跟踪节点名称监听地址，在集群中起到一个类似DNS的作用。
- 5672, 5671：AMQP 0-9-1 和 1.0 客户端端口，没有使用SSL和使用SSL的端口。
- 25672：用于RabbitMQ节点间和CLI工具通信，配合4369使用。
- 15672：HTTP_API端口，管理员用户才能访问，用于管理RbbitMQ，需要启用management插件。
- 61613,61614：当STOMP插件启用的时候打开，作为STOMP客户端端口（根据是否使用TLS选择）。
- 1883, 8883：当MQTT插件启用的时候打开，作为MQTT客户端端口（根据是否使用TLS选择）。
- 15674：基于WebSocket的STOMP客户端端口（当插件Web STOMP启用的时候打开）
- 15675：基于WebSocket的MQTT客户端端口（当插件Web MQTT启用的时候打开）

RabbitMQ管理界面

RabbitMQ 安装包中带有管理插件，但需要手动激活

```
rabbitmq-plugins enable rabbitmq_management
```

RabbitMQ 有一个默认的用户“guest”，但这个用户默认只能通过本机访问，要让其它机器可以访问，需要创建一个新用户，为其分配权限

```
#添加用户
rabbitmqctl add_user admin admin
#为用户分配权限
rabbitmqctl set_user_tags admin administrator
#为用户分配资源权限
rabbitmqctl set_permissions -p / admin ".*" ".*" ".*"
```

RabbitMQ 角色分类

RabbitMQ 的用户角色分类：none、management、policymaker、monitoring、administrator

- 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

hello world

RabbitMQ 支持多种语言访问，以 Java 为例看下一般使用 RabbitMQ 的步骤。

1、maven工程的pom文件中添加依赖

```
<dependency>
    <groupId>com.rabbitmq</groupId>
    <artifactId>amqp-client</artifactId>
    <version>4.1.0</version>
</dependency>
```

2、消息生产者

```
package org.gupaoedu.rabbitmq;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;
import java.io.IOException;
import java.util.concurrent.TimeoutException;
public class Producer {

    public static void main(String[] args) throws IOException,
TimeoutException {
        //创建连接工厂
        ConnectionFactory factory = new ConnectionFactory();
        factory.setUsername("guest");
        factory.setPassword("guest");
        //设置 RabbitMQ 地址
```

```
        factory.setHost("localhost");
        //建立到代理服务器到连接
        Connection conn = factory.newConnection();
        //获得信道
        Channel channel = conn.createChannel();
        //声明交换器
        String exchangeName = "hello-exchange";
        channel.exchangeDeclare(exchangeName, "direct", true);

        String routingKey = "hola";
        //发布消息
        byte[] messageBodyBytes = "quit".getBytes();
        channel.basicPublish(exchangeName, routingKey, null,
messageBodyBytes);

        channel.close();
        conn.close();
    }

}
```

3、消息消费者

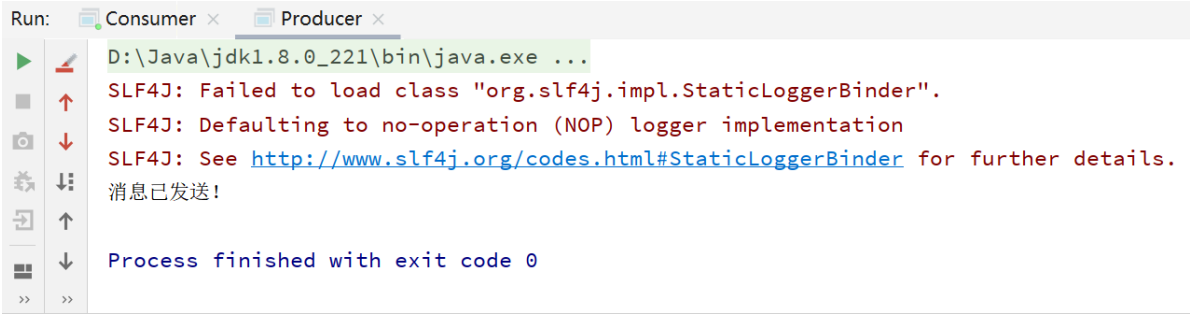
```
package org.gupaoedu.rabbitmq;
import com.rabbitmq.client.*;
import java.io.IOException;
import java.util.concurrent.TimeoutException;
public class Consumer {

    public static void main(String[] args) throws IOException,
TimeoutException {
        ConnectionFactory factory = new ConnectionFactory();
        factory.setUsername("guest");
        factory.setPassword("guest");
        factory.setHost("localhost");
        //建立到代理服务器到连接
        Connection conn = factory.newConnection();
        //获得信道
        final Channel channel = conn.createChannel();
        //声明交换器
        String exchangeName = "hello-exchange";
        channel.exchangeDeclare(exchangeName, "direct", true);
        //声明队列
        String queueName = channel.queueDeclare().getQueue();
        String routingKey = "hola";
        //绑定队列，通过键 hola 将队列和交换器绑定起来
        channel.queueBind(queueName, exchangeName, routingKey);

        while(true) {
            //消费消息
            boolean autoAck = false;
            String consumerTag = "";
            channel.basicConsume(queueName, autoAck, consumerTag, new
DefaultConsumer(channel) {
                @Override
                public void handleDelivery(String consumerTag,
                                           Envelope envelope,
```

```
                                           AMQP.BasicProperties properties,
                                           byte[] body) throws IOException {
                    String routingKey = envelope.getRoutingKey();
                    String contentType = properties.getContentType();
                    System.out.println("消费的路由键：" + routingKey);
                    System.out.println("消费的内容类型：" + contentType);
                    long deliveryTag = envelope.getDeliveryTag();
                    //确认消息
                    channel.basicAck(deliveryTag, false);
                    System.out.println("消费的消息体内容：");
                    String bodyStr = new String(body, "UTF-8");
                    System.out.println(bodyStr);

                }
            });
        }
    }

}
```

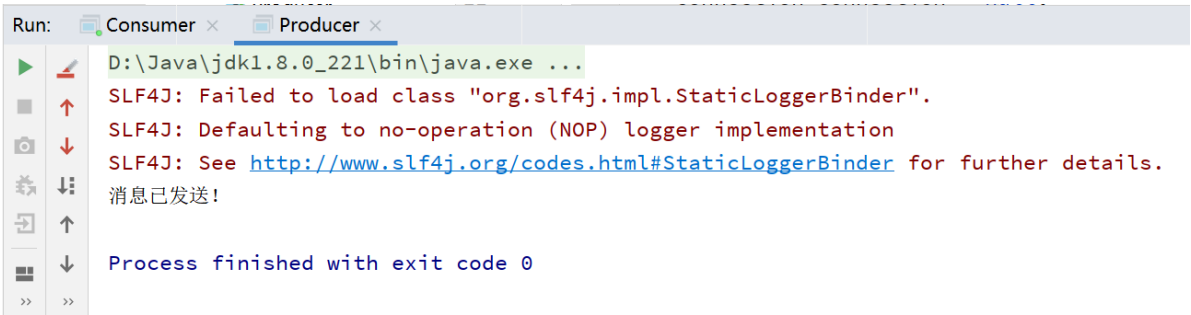
4、启动 RabbitMQ 服务器

```
./sbin/rabbitmq-server
```

5、运行 Consumer
先运行 Consumer ，这样当生产者发送消息的时候能在消费者后端看到消息记录。



6、运行 Producer
接着运行 Producer ,发布一条消息，在 Consumer 的控制台能看到接收的消息：



AMQP 协议

AMQP （ Advanced Message Queuing Protocol）高级消息队列协议，是应用层协议的一个开放标准，为面向消息的中间件设计。

- AMQP结构

- Module Layer：位于协议最高层，主要定义了一些供客户端调用的命令，客户端可以利用这些命令实现自己的业务逻辑。
  例如，客户端可以使用 Queue.Declare 命令声明一个队列或者使用 Basic.Consume 订阅消费一个队列中的消息。
- Session Layer：位于中间层，主要负责将客户端的命令发送给服务器，再将服务端的应答返回给客户端，主要为客户端与服务器之间的通信提供可靠性同步机制和错误处理。
- Transport Layer：位于最底层，主要传输二进制数据流 ，提供帧的处理、信道复用、错误检测和数据表示等。

AMQP 说到底还是一个通信协议，通信协议都会涉及报文交互
从 low-level 举例来说，AMQP 本身是应用层的协议，其填充于 TCP 协议层的数据部分。
从 high-level 来说， AMQP是通过协议命令进行交互的。
AMQP 协议可以看作一系列结构化命令的集合，这里的命令代表一种操作，类似于 HTTP 中的方法(GET、 POST 、 PUT、 DELETE 等) 。

- AMQP 生产者流转过程

  当客户端与 Broker 建立连接的时候，会调用 factory.newConnection 方法，
  这个方法会进一步封装成 Protocol Header 0-9-1 的报文头发送给 Broker，
  以此通知 Broker 本次交互采用的是 AMQP0-9-1 协议，
  紧接着 Broker 返回 Connection.Start 来建立连接，
  在连接的过程中涉及 Connection.Start/.Start-OK 、 Connection.Tune/.Tune-Ok ，
  Connection.Open/ .Open-Ok 这 6 个命令的交互 。

  当客户端调用 connection.createChannel 方法准备开启信道的时候，其包装Channel.Open 命令发送给 Broker，等待 Channel.Open-Ok 命令 。

  当客户端发送消息的时候，需要调用 channel.basicPublish 方法，对应的 AQMP 命令为 Basic.Publish ，
  注意这个命令和前面涉及的命令略有不同，这个命令还包含了 ContentHeader 和 Content Body。
  Content Header 里面包含的是消息体的属性，例如，投递模式 、优先级等。
  而 Content Body 包含消息体本身。

  当客户端发送完消息需要关闭资源时，涉及 Channel.Close/.Close-Ok 与 Connection.Close/.Close-Ok 的命令交互

- AMQP 消费者流转过程

  消费者客户端同样需要与Broker 建立连接
  与生产者客户端一样，协议交互同样涉及Connection.Start/ . Start-Ok 、 Connection.Tune/.Tune-Ok 和 Connection.Open/ . Open-Ok 等，这里中省略了这些步骤
  紧接着也少不了在 Connection 之上建立 Channel，和生产者客户端一样，协议涉及Channel . Open/Open-Ok。

  如果在消费之前调用了 channel.basicQos(int prefetchCount) 的方法来设置消费者客户端最大能“保持”的未确认的消息数，那么协议流转会涉及 Basic.Qos/.Qos-Ok 这两个 AMQP 命令。

  在真正消费之前，消费者客户端需要向 Broker 发送 Basic.Consume 命令(即调用 channel.basicConsume 方法〉将 Channel 置为接收模式，之后 Broker 回执Basic.Consume-Ok 以告诉消费者客户端准备好消费消息。紧接着 Broker 向消费者客户端推送 (Push) 消息，即 Basic.Deliver 命令，有意思的是这个和 Basic.Publish 命令一样会携带 Content Header 和 Content Body 。

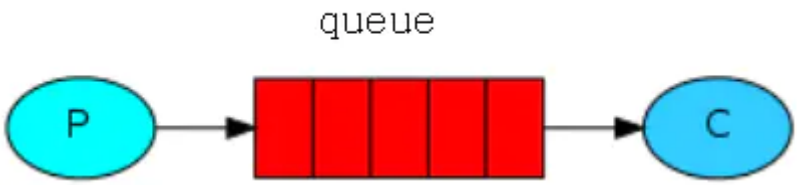
  消费者接收到消息并正确消费之后，向 Broker 发送确认，即 Basic.Ack 命令。

  在消费者停止消费的时候，主动关闭连接，这点和生产者 一 样，涉及Channel . Close/ . Close-Ok 手口 Connection.Close/ . Close-Ok 。
  AMQP还有很多其它命令，不在此介绍

RabbitMQ中的概念模型

消息模型

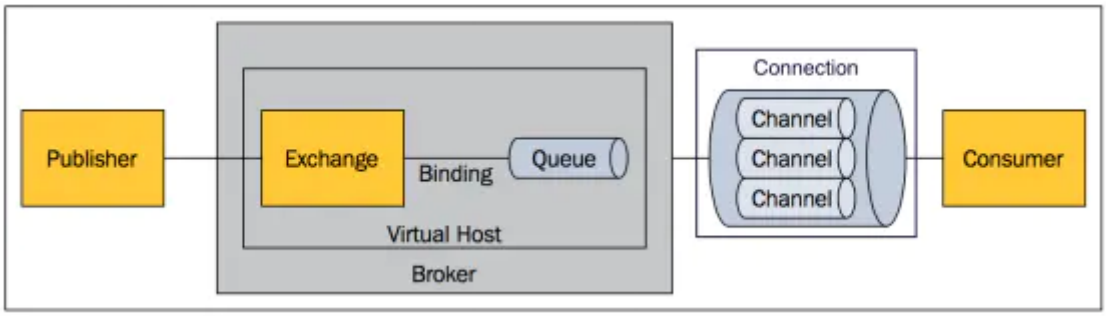
所有 MQ 产品从模型抽象上来说都是一样的过程：
消费者（consumer）订阅某个队列。生产者（producer）创建消息，然后发布到队列（queue）中，最后将消息发送到监听的消费者。



消息流

RabbitMQ 基本概念

上面只是最简单抽象的描述，具体到 RabbitMQ 则有更详细的概念需要解释。上面介绍过 RabbitMQ 是 AMQP 协议的一个开源实现，所以其内部实际上也是 AMQP 中的基本概念：



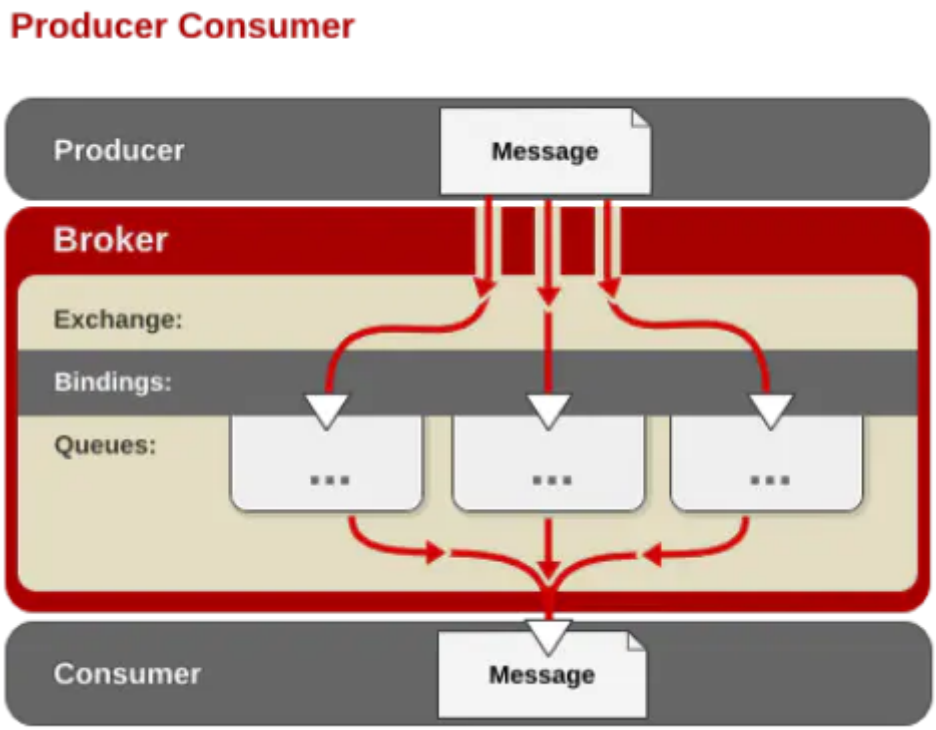
RabbitMQ 内部结构

1. Message
   消息，消息是不具名的，它由消息头和消息体组成。消息体是不透明的，而消息头则由一系列的可选属性组成，这些属性包括routing-key（路由键）、priority（相对于其他消息的优先权）、delivery-mode（指出该消息可能需要持久性存储）等。
2. Publisher
   消息的生产者，也是一个向交换器发布消息的客户端应用程序。
3. Exchange
   交换器，用来接收生产者发送的消息并将这些消息路由给服务器中的队列。
4. Binding
   绑定，用于消息队列和交换器之间的关联。一个绑定就是基于路由键将交换器和消息队列连接起来的路由规则，所以可以将交换器理解成一个由绑定构成的路由表。
5. Queue
   消息队列，用来保存消息直到发送给消费者。它是消息的容器，也是消息的终点。一个消息可投入一个或多个队列。消息一直在队列里面，等待消费者连接到这个队列将其取走。
6. Connection
   网络连接，比如一个TCP连接。
7. Channel
   信道，多路复用连接中的一条独立的双向数据流通道。信道是建立在真实的TCP连接内地虚拟连接，AMQP 命令都是通过信道发出去的，不管是发布消息、订阅队列还是接收消息，这些动作都是通过信道完成。因为对于操作系统来说建立和销毁 TCP 都是非常昂贵的开销，所以引入了信道的概念，以复用一条 TCP 连接。

8. Consumer
   消息的消费者，表示一个从消息队列中取得消息的客户端应用程序。
9. Virtual Host
   虚拟主机，表示一批交换器、消息队列和相关对象。虚拟主机是共享相同的身份认证和加密环境的独立服务器域。每个 vhost 本质上就是一个 mini 版的 RabbitMQ 服务器，拥有自己的队列、交换器、绑定和权限机制。vhost 是 AMQP 概念的基础，必须在连接时指定，RabbitMQ 默认的 vhost 是 / 。
10. Broker
    表示消息队列服务器实体。

AMQP 中的消息路由

AMQP 中消息的路由过程和 Java 开发者熟悉的 JMS 存在一些差别，AMQP 中增加了 Exchange 和 Binding 的角色。生产者把消息发布到 Exchange 上，消息最终到达队列并被消费者接收，而 Binding 决定交换器的消息应该发送到那个队列。

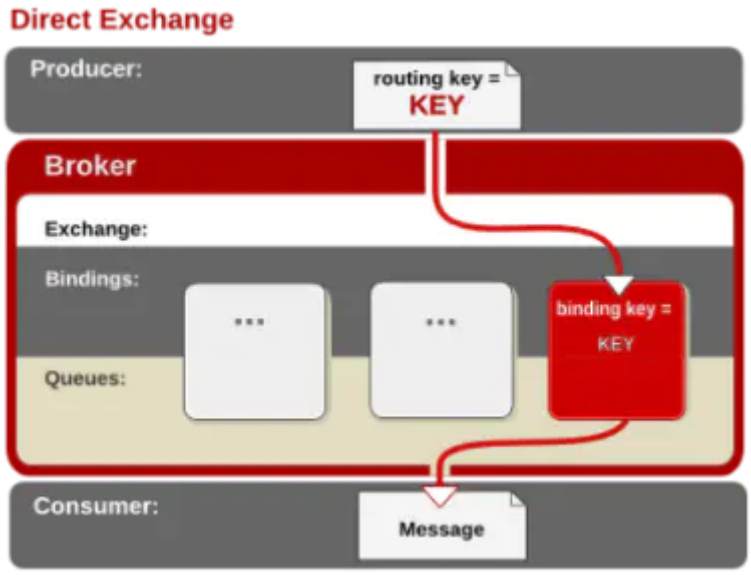


AMQP 的消息路由过程

Exchange 类型

Exchange分发消息时根据类型的不同分发策略有区别，目前共四种类型：direct、fanout、topic、headers 。headers 匹配 AMQP 消息的 header 而不是路由键，此外 headers 交换器和 direct 交换器完全一致，但性能差很多，目前几乎用不到了，所以直接看另外三种类型：

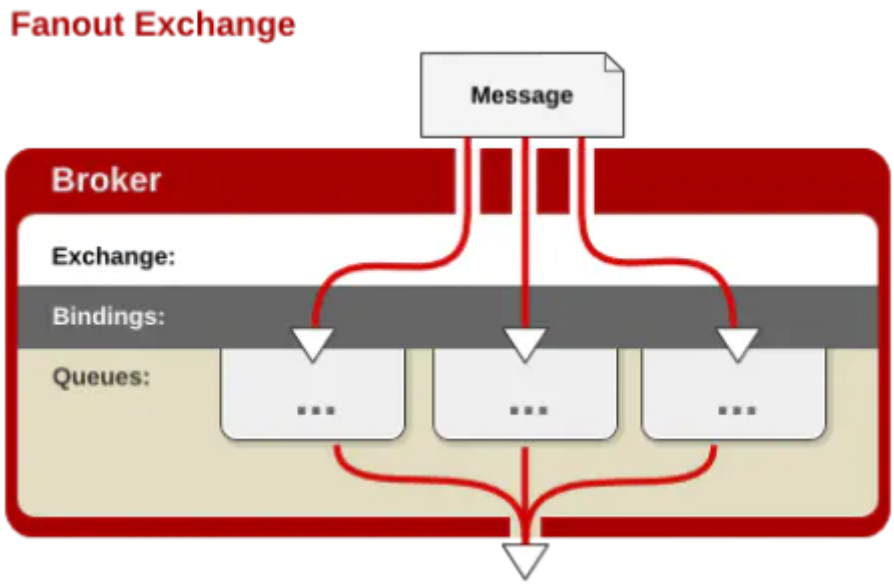
1、direct



direct 交换器

消息中的路由键（routing key）如果和 Binding 中的 binding key 一致， 交换器就将消息发到对应的队列中。路由键与队列名完全匹配，如果一个队列绑定到交换机要求路由键为“dog”，则只转发 routing key 标记为“dog”的消息，不会转发“dog.puppy”，也不会转发“dog.guard”等等。它是完全匹配、单播的模式。

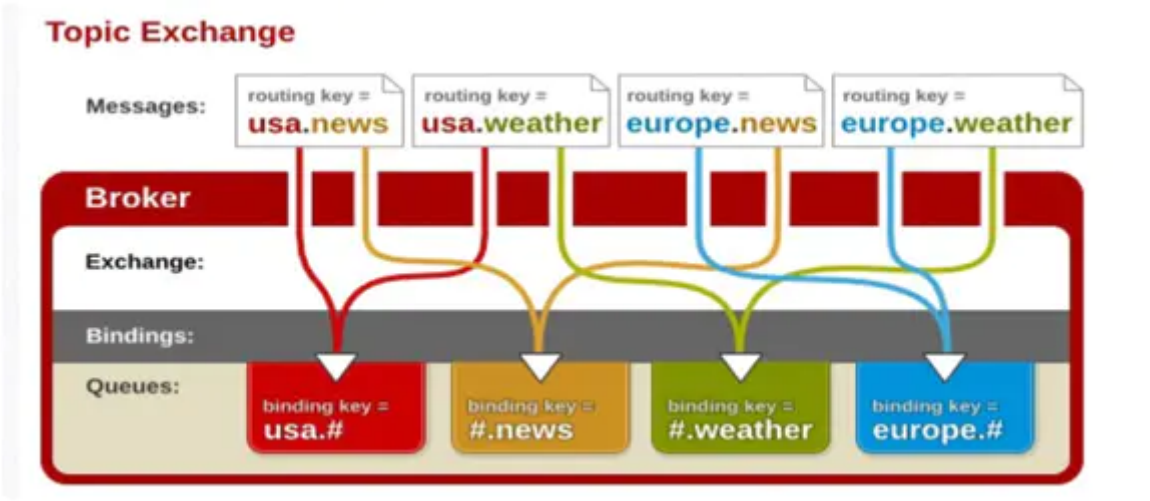
2、fanout



fanout 交换器

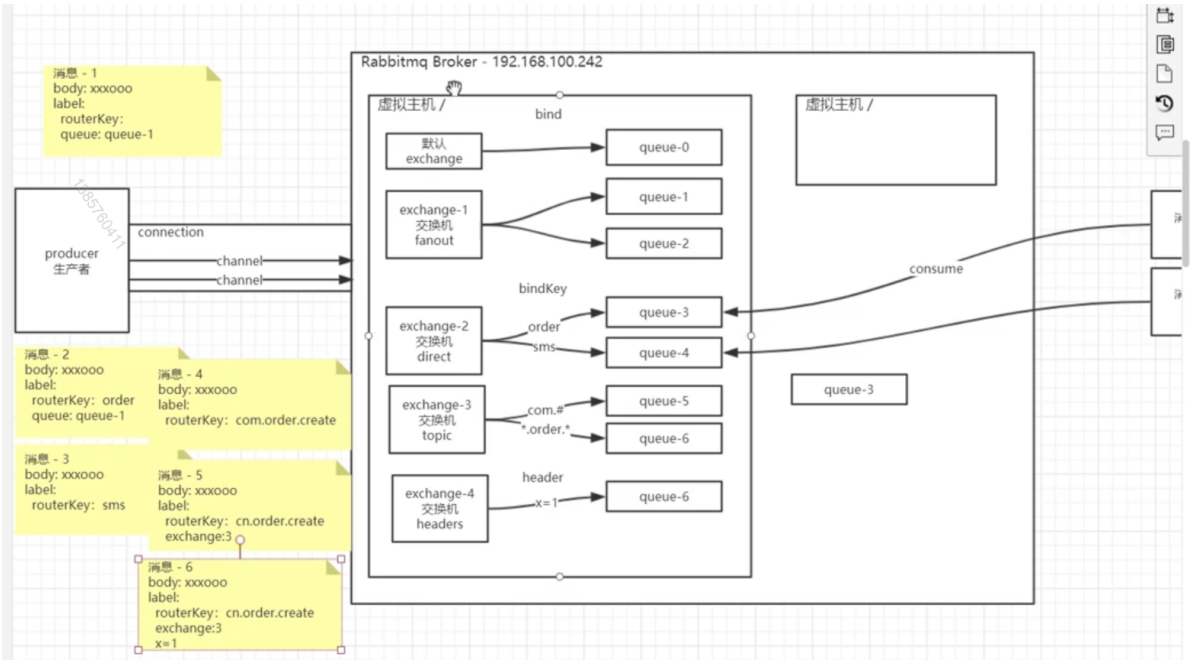
每个发到 fanout 类型交换器的消息都会分到所有绑定的队列上去。fanout 交换器不处理路由键，只是简单的将队列绑定到交换器上，每个发送到交换器的消息都会被转发到与该交换器绑定的所有队列上。很像子网广播，每台子网内的主机都获得了一份复制的消息。fanout 类型转发消息是最快的。

3、topic



topic 交换器

topic 交换器通过模式匹配分配消息的路由键属性，将路由键和某个模式进行匹配，此时队列需要绑定到一个模式上。它将路由键和绑定键的字符串切分成单词，这些单词之间用点隔开。它同样也会识别两个通配符：符号“#”和符号“”。*#匹配0个或多个单词，*匹配不多不少一个单词。



RabbitMQ 集群部署方式

多机多节点集群部署

一、 环境准备

- 准备三台安装好RabbitMQ 的机器，安装方法见 安装步骤
  - 10.10.1.41
  - 10.10.1.42
  - 10.10.1.43

`提示：如果使用虚拟机，可以在一台VM上安装好RabbitMQ后，创建快照，从快照创建链接克隆，会节省很多磁盘空间`

二、修改配置文件

1. 修改`10.10.1.41`机器上的`/etc/hosts`文件

```
sudo vim /etc/hosts
```

2. 添加IP和节点名

```
10.10.1.41 node1
10.10.1.42 node2
10.10.1.43 node3
```

3. 修改对应主机的hostname

```
hostname node1
hostname node2
hostname node3
```

4. 将`10.10.1.41`上的hosts文件复制到另外两台机器上

```
sudo scp /etc/hosts root@node2:/etc/
sudo scp /etc/hosts root@node3:/etc/
```

`说明：命令中的root是目标机器的用户名，命令执行后，可能会提示需要输入密码，输入对应用户的密码就行了`

5. 将`10.10.1.41`上的`/var/lib/rabbitmq/.erlang.cookie`文件复制到另外两台机器上

```
scp /var/lib/rabbitmq/.erlang.cookie root@node2:/var/lib/rabbitmq/
scp /var/lib/rabbitmq/.erlang.cookie root@node3:/var/lib/rabbitmq/
```

`提示：如果是通过克隆的VM，可以省略这一步`

三、防火墙添加端口

- 给每台机器的防火墙添加端口

1. 添加端口

```
sudo firewall-cmd --zone=public --add-port=4369/tcp --permanent
sudo firewall-cmd --zone=public --add-port=5672/tcp --permanent
sudo firewall-cmd --zone=public --add-port=25672/tcp --permanent
sudo firewall-cmd --zone=public --add-port=15672/tcp --permanent
```

2. 重启防火墙

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

四、启动RabbitMQ

1. 启动每台机器的RabbitMQ

```
sudo systemctl start rabbitmq-server
```

或者

```
rabbitmq-server -detached
```

2. 将`10.10.1.42`加入到集群

```
# 停止RabbitMQ 应用
rabbitmqctl stop_app
# 重置RabbitMQ 设置
rabbitmqctl reset
# 加入到集群
rabbitmqctl join_cluster rabbit@node1 --ram
# 启动RabbitMQ 应用
rabbitmqctl start_app
```

3. 查看集群状态，看到`running_nodes,[rabbit@node1,rabbit@node2]`表示节点启动成功

```
rabbitmqctl cluster_status
```

`提示：在管理界面可以更直观的看到集群信息`

4. 将`10.10.1.43`加入到集群

```
# 停止 RabbitMQ 应用
rabbitmqctl stop_app
# 重置 RabbitMQ 设置
rabbitmqctl reset
# 节点加入到集群
rabbitmqctl join_cluster rabbit@node1 --ram
# 启动 RabbitMQ 应用
rabbitmqctl start_app
```

5. 重复地3步，查看集群状态

单机多节点部署

一、环境准备

- 准备一台已经安装好RabbitMQ 的机器，安装方法见 安装步骤
  - 10.10.1.41

二、防火墙添加端口

- 需要将每个节点的端口都添加到防火墙

1. 添加端口

```
sudo firewall-cmd --zone=public --add-port=4369/tcp --permanent
sudo firewall-cmd --zone=public --add-port=5672/tcp --permanent
sudo firewall-cmd --zone=public --add-port=25672/tcp --permanent
sudo firewall-cmd --zone=public --add-port=15672/tcp --permanent
sudo firewall-cmd --zone=public --add-port=5673/tcp --permanent
sudo firewall-cmd --zone=public --add-port=25673/tcp --permanent
sudo firewall-cmd --zone=public --add-port=15673/tcp --permanent
sudo firewall-cmd --zone=public --add-port=5674/tcp --permanent
sudo firewall-cmd --zone=public --add-port=25674/tcp --permanent
sudo firewall-cmd --zone=public --add-port=15674/tcp --permanent
```

2. 重启防火墙

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

三、启动RabbitMQ

1. 在启动前，先修改RabbitMQ 的默认节点名（非必要），在`/etc/rabbitmq/rabbitmq-env.conf`增加以下内容

```
# RabbitMQ 默认节点名，默认是rabbit
RABBITMQ_NODE_PORT=5672 RABBITMQ_NODENAME=rabbit1
```

2. RabbitMQ 默认是使用服务的启动的，单机多节点时需要改为手动启动，先停止运行中的RabbitMQ 服务

```
sudo systemctl stop rabbitmq-server
```

3. 启动第一个节点

```
rabbitmq-server -detached
```

4. 启动第二个节点

```
RABBITMQ_NODE_PORT=5673 RABBITMQ_SERVER_START_ARGS="-rabbitmq_management
listener [{port,15673}]" RABBITMQ_NODENAME=rabbit2 rabbitmq-server -
detached
```

5. 启动第三个节点

```
RABBITMQ_NODE_PORT=5674 RABBITMQ_SERVER_START_ARGS="-rabbitmq_management
listener [{port,15674}]" RABBITMQ_NODENAME=rabbit3 rabbitmq-server -
detached
```

6. 将rabbit2加入到集群

```
# 停止 rabbit2 的应用
rabbitmqctl -n rabbit2 stop_app
# 重置 rabbit2 的设置
rabbitmqctl -n rabbit2 reset
# rabbit2 节点加入到 rabbit1的集群中
rabbitmqctl -n rabbit2 join_cluster rabbit1 --ram
# 启动 rabbit2 节点
rabbitmqctl -n rabbit2 start_app
```

7. 将rabbit3加入到集群

```
# 停止 rabbit3 的应用
rabbitmqctl -n rabbit3 stop_app
# 重置 rabbit3 的设置
rabbitmqctl -n rabbit3 reset
# rabbit3 节点加入到 rabbit1的集群中
rabbitmqctl -n rabbit3 join_cluster rabbit1 --ram
# 启动 rabbit3 节点
rabbitmqctl -n rabbit3 start_app
```

8. 查看集群状态，看到`{running_nodes,[rabbit3@node1,rabbit2@node1,rabbit1@node1]}`说明节点已启动成功。

```
rabbitmqctl cluster_status
```

`提示：在管理界面可以更直观的看到集群信息`

9. 移除节点

```
首先将要移除的节点停机
rabbitmqctl -n rabbit3 stop_app
在主节点,也就是发起进群的主机上进行节点的移除
rabbitmqctl -n rabbit forget_cluster_node rabbit3
```

镜像队列模式集群

- 镜像队列属于RabbitMQ 的高可用方案，见：https://www.rabbitmq.com/ha.html#mirroring-arguments
- 通过前面的步骤搭建的集群属于普通模式集群，是通过共享元数据实现集群
- 开启镜像队列模式需要在管理页面添加策略，添加方式：
    1. 进入管理页面 -> Admin -> Policies（在页面右侧）-> Add / update a policy
    2. 在表单中填入：

```
name: ha-all
Pattern: ^
Apply to: Queues
Priority: 0
Definition: ha-mode = all
```

参数说明

name: 策略名称，如果使用已有的名称，保存后将会修改原来的信息

Apply to：策略应用到什么对象上

Pattern：策略应用到对象时，对象名称的匹配规则（正则表达式）

Priority：优先级，数值越大，优先级越高，相同优先级取最后一个

Definition：策略定义的类容，对于镜像队列的配置来说，只需要包含3个部分: `ha-mode` 、`ha-params` 和 `ha-sync-mode`。其中，`ha-sync-mode`是同步的方式，自动还是手动，默认是自动。`ha-mode` 和 `ha-params` 组合使用。组合方式如下：

ha-mode	ha-params	说明
all	(empty)	队列镜像到集群类所有节点
exactly	count	队列镜像到集群内指定数量的节点。如果集群内节点数少于此值，队列将会镜像到所有节点。如果大于此值，而且一个包含镜像的节点停止，则新的镜像不会在其它节点创建。
nodes	nodename	队列镜像到指定节点，指定的节点不在集群中不会报错。当队列申明时，如果指定的节点不在线，则队列会被创建在客户端所连接的节点上。

- 镜像队列模式相比较普通模式，镜像模式会占用更多的带宽来进行同步，所以镜像队列的吞吐量会低于普通模式
- 但普通模式不能实现高可用，某个节点挂了后，这个节点上的消息将无法被消费，需要等待节点启动后才能被消费。

示例说明

此示例演示Virtual Hosts和权限的使用，及客户端链接集群的用法。

Virtual Hosts

每一个 RabbitMQ 服务器都能创建虚拟的消息服务器，我们称之为虚拟主机 (virtual host) ,简称为 vhost。

每一个 vhost 本质上是一个独立的小型 RabbitMQ 服务器，拥有自己独立的队列、交换器及绑定关系等，并且它拥有自己独立的权限。

vhost 就像是虚拟机与物理服务器一样，它们在各个实例间提供逻辑上的分离，为不同程序安全保密地运行数据，它既能将同一个RabbitMQ 中的众多客户区分开，又可以避免队列和交换器等命名冲突。

vhost 之间是绝对隔离的，无法将 vhostl 中的交换器与 vhost2 中的队列进行绑定，这样既保证了安全性，又可以确保可移植性。

如果在使用 RabbitMQ 达到一定规模的时候，建议用户对业务功能、场景进行归类区分，并为之分配独立的 vhost。

Virtual Hosts 的功能说明

vhost可以限制最大连接数和最大队列数，并且可以设置vhost下的用户资源权限和Topic权限，具体权限见下方说明。

- 在 `Admin -> Limits` 页面可以设置vhost的最大连接数和最大队列数，达到限制后，继续创建，将会报错。
- 用户资源权限是指RabbitMQ 用户在客户端执行AMQP操作命令时，拥有对资源的操作和使用权限。权限分为三个部分：`configure、write、read`，见下方表格说明。参考：http://www.rabbitmq.com/access-control.html#permissions

AMQP 0-9-1 Operation		configure	write	read
exchange.declare	(passive=false)	exchange		
exchange.declare	(passive=true)			
exchange.declare	(with AE)	exchange	exchange (AE)	exchange
exchange.delete		exchange		
queue.declare	(passive=false)	queue		
queue.declare	(passive=true)			
queue.declare	(with DLX)	queue	exchange (DLX)	queue
queue.delete		queue		
exchange.bind			exchange (destination)	exchange (source)
exchange.unbind			exchange (destination)	exchange (source)
queue.bind			queue	exchange
queue.unbind			queue	exchange
basic.publish			exchange	
basic.get				queue
basic.consume				queue
queue.purge				queue

举例说明：

  - 比如创建队列时，会调用`queue.declare`方法，此时会使用到`configure`权限，会校验队列名是否与`configure`的表达式匹配。
  - 比如队列绑定交换器时，会调用`queue.bind`方法，此时会用到`write`和`read`权限，会检验队列名是否与`write`的表达式匹配，交换器名是否与`read`的表达式匹配。
- Topic权限，参考：http://www.rabbitmq.com/access-control.html#topic-authorisation
  - Topic权限是RabbitMQ 针对STOMP和MQTT等协议实现的一种权限。由于这类协议都是基于Topic消费的，而AMQP是基于Queue消费，所以AMQP的标准资源权限不适合用在这类协议

中，而Topic权限也不适用于AMQP协议。所以，我们一般不会去使用它，只用在使用了MQTT这类的协议时才可能会用到。

vhost使用示例

1. 使用管理员用户登录Web管理界面。
2. 在 `Admin -> Virtual Hosts` 页面添加一个名为 `v1` 的Virtual Hosts。
    - 此时还需要为此vhost分配用户，添加一个新用户
3. 在 `Admin -> Users` 页面添加一个名为 `order-user` 的用户，并设置为 `management` 角色。
4. 从 `Admin` 进入 `order-user` 的用户设置界面,在 `Permissions` 中，为用户分配vhost为/v1，并为每种权限设置需要匹配的目标名称的正则表达式。

字段名	值	说明
Virtual Host	/v1	指定用户的vhost，以下权限都只限于 `/v1` vhost中
Configure regexp	eq-.*	只能操作名称以eq-开头的exchange或queue；为空则不能操作任何exchange和queue
Write regexp	.*	能够发送消息到任意名称的exchange，并且能绑定到任意名称的队列和任意名称的目标交换器（指交换器绑定到交换器），为空表示没有权限
Read regexp	^test$	只能消费名为test队列上的消息，并且只能绑定到名为test的交换器

5. 执行示例代码 `VirtualHostsDemo`。

集群连接恢复

- **参考**：https://www.rabbitmq.com/api-guide.html#connection-recovery
- 通过 `factory.setAutomaticRecoveryEnabled(true);` 可以设置连接自动恢复的开关，默认已开启
- 通过 `factory.setNetworkRecoveryInterval(10000);` 可以设置间隔多长时间尝试恢复一次，默认是5秒：`com.rabbitmq.client.ConnectionFactory.DEFAULT_NETWORK_RECOVERY_INTERVAL`
- 什么时候会触发连接恢复？https://www.rabbitmq.com/api-guide.html#recovery-triggers
    - 如果启用了自动连接恢复，将由以下事件触发：
        - 连接的I/O循环中抛出IOExceiption
        - 读取Socket套接字超时
        - 检测不到服务器心跳
        - 在连接的I/O循环中引发任何其他异常
    - 如果客户端第一次连接失败，不会自动恢复连接。需要我们自己负责重试连接、记录失败的尝试、实现重试次数的限制等等。
      ```java
      ConnectionFactory factory = new ConnectionFactory();
      // 设置连接配置
      ```

```
try {
    Connection conn = factory.newConnection();
} catch (java.net.ConnectException e) {
    Thread.sleep(5000);
    // 重新连接
}
```

  - 如果程序中调用了`Connection.Close`，也不会自动恢复连接。
  - 如果是`Channel-level`的异常，也不会自动恢复连接，因为这些异常通常是应用程序中存在语义问题(例如试图从不存在的队列消费)。
- 在Connection和Channel上，可以设置重新连接的监听器，开始重连和重连成功时，会触发监听器。添加和移除监听，需要将Connection或Channel强制转换成Recoverable接口。

```
((Recoverable) connection).addRecoveryListener()
((Recoverable) connection).removeRecoveryListener()
```

重连测试方式

- 测试前，先按集群部署方式搭建好集群。
- 开启集群节点后，启动`Consumer`和`Producer`。
- 使用`rabbitmqctl -n [node_name] stop_app`命令关闭一个节点，例如：`rabbitmqctl -n rabbit2 stop_app`;
- 查看Consumer和Producer控制台是否有重连的信息。
- 使用`rabbitmqctl -n [node_name] start_app`可开启关闭的节点。

镜像队列测试

- 测试方式
  - 生产者连接10.10.1.41:5672发送消息后，停止rabbit1节点

队列持久化	消息持久化	镜像队列	结果
否	否	否	rabbit1重启后，队列和消息丢失
是	否	否	rabbit1重启后，队列存在但消息丢失；rabbit1不启动消费者连接其它节点也无法启动
是	是	否	rabbit1重启后，队列存在，消息丢失；rabbit1不启动消费者连接其它节点也无法启动
否	否	是	对列和消息都还存在，并且消费者能够正常消费
是	否	是	同上
是	是	是	同上

RabbitMQ持久化机制、内存磁盘控制

示例说明

此示例演示交换器、队列、消息持久化功能。和内存、磁盘预警

持久化

RabbitMQ 的持久化分交换器持久化、队列持久化和消息持久化。

- 定义持久化交换器，通过第三个参数 `durable` 开启/关闭持久化

```
channel.exchangeDeclare(exchangeName, exchangeType, durable)
```

- 定义持久化队列，通过第二个参数 `durable` 开启/关闭持久化

```
channel.queueDeclare(queue, durable, exclusive, autoDelete, arguments);
```

- 发送持久化消息，需要在消息属性中设置 `deliveryMode=2`， 此属性在 `BasicProperties` 中，通过 `basicPublish` 方法的 `props` 参数传入。

```
channel.basicPublish(exchange, routingKey, props, body);
```

`BasicProperties` 对象可以从RabbitMQ 内置的 `MessageProperties` 类中获取

```
MessageProperties.PERSISTENT_TEXT_PLAIN
```

如果还需要设置其它属性，可以通过 `AMQP.BasicProperties.Builder` 去构建一个 `BasicProperties` 对象

```
new AMQP.BasicProperties.Builder()
        .deliveryMode(2)
        .build()
```

内存告警

默认情况下 `set_vm_memory_high_watermark` 的值为 0.4，即内存阈值（临界值）为 0.4，表示当 RabbitMQ 使用的内存超过 40%时，就会产生内存告警并阻塞所有生产者的连接。一旦告警被解除(有消息被消费或者从内存转储到磁盘等情况的发生)， 一切都会恢复正常。

在出现内存告警后，所有的客户端连接都会被阻塞。阻塞分为 `blocking` 和 `blocked` 两种。

- blocking：表示没有发送消息的链接。
- blocked：表示试图发送消息的链接。

如果出现了内存告警，并且机器还有可用内存，可以通过命令调整内存阈值，解除告警。

```
rabbitmqctl set_vm_memory_high_watermark 1
```

或者

```
rabbitmqctl set_vm_memory_high_watermark absolute 1GB
```

但这种方式只是临时调整，RabbitMQ 服务重启后，会还原。如果需要永久调整，可以修改配置文件。但修改配置文件需要**重启RabbitMQ 服务才能生效。**

- 修改配置文件：`vim /etc/rabbitmq/rabbitmq.conf`

  ```
  vm_memory_high_watermark.relative = 0.4
  ```

  或者

  ```
  vm_memory_high_watermark.absolute = 1GB
  ```

模拟内存告警

1. 调整内存阈值，模拟出告警，在RabbitMQ 服务器上修改。`注意：修改之前，先在管理页面看一下当前使用了多少，调成比当前值小`

   ```
   rabbitmqctl set_vm_memory_high_watermark absolute 50MB
   ```

2. 刷新管理页面（可能需要刷新多次），在 `Overview -> Nodes` 中可以看到Memory变成了红色，表示此节点内存告警了
3. 启动 `Producer` 和 `Consumer`
4. 查看管理界面的 `Connections` 页面，可以看到生产者和消费者的链接都处于 `blocking` 状态。
5. 在 `Producer` 的控制台按回车健，再观察管理界面的 `Connections` 页面，会发现生产者的状态成了 `blocked`。
6. 此时虽然在`Producer` 控制台看到了发送两条消息的信息，但 `Consumer` 并没有收到任何消息。并且在管理界面的 `Queues` 页面也看到不到队列的消息数量有变化。
7. 解除内存告警后，会发现 `Consumer` 收到了 `Producer` 发送的两条消息。

内存换页

- 在Broker节点的使用内存即将达到内存阈值之前，它会尝试将队列中的消息存储到磁盘以释放内存空间，这个动作叫内存换页。
- 持久化和非持久化的消息都会被转储到磁盘中，其中持久化的消息本身就在磁盘中有一份副本，此时会将持久化的消息从内存中清除掉。
- 默认情况下，在内存到达内存阈值的 50%时会进行换页动作。也就是说，在默认的内存阈值为 0.4 的情况下，当内存超过 0.4 x 0 .5=0.2 时会进行换页动作。
- 通过修改配置文件，调整内存换页分页阈值（不能通过命令调整）。

  ```
  # 此值大于1时，相当于禁用了换页功能。
  vm_memory_high_watermark_paging_ratio = 0.75
  ```

磁盘告警

- 当磁盘剩余空间低于磁盘的阈值时，RabbitMQ 同样会阻塞生产者，这样可以避免因非持久化的消息持续换页而耗尽磁盘空间导致服务崩溃
- 默认情况下，磁盘阈值为50MB，表示当磁盘剩余空间低于50MB 时会阻塞生产者并停止内存中消息的换页动作
- 这个阈值的设置可以减小，但不能完全消除因磁盘耗尽而导致崩溃的可能性。比如在两次磁盘空间检测期间内，磁盘空间从大于50MB被耗尽到0MB
- 通过命令可以调整磁盘阈值，临时生效，重启恢复

```
# disk_limit 为固定大小，单位为MB、GB
rabbitmqctl set_disk_free_limit <disk_limit>
```

或者

```
# fraction 为相对比值，建议的取值为1.0~2.0之间
rabbitmqctl set_disk_free_limit mem_relative <fraction>
```

模拟磁盘告警

1. 在服务器通过命令，临时调整磁盘阈值（**需要设置一个绝对大与当前磁盘空间的数值**）

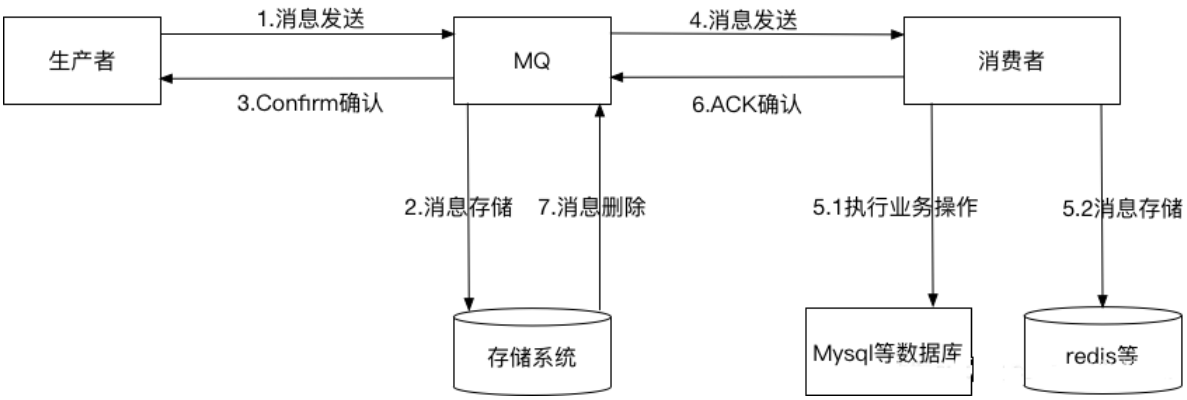
```
rabbitmqctl set_disk_free_limit 102400GB
```

2. 刷新管理页面（可能需要刷新多次），在 Overview -> Nodes 中可以看到Disk space变成了红色，表示此节点磁盘告警了
3. 后续步骤同模拟内存告警。

消息可靠性

RabbitMQ消息的可靠性投递主要两种实现：
1、通过实现消费的重试机制，通过@Retryable来实现重试，可以设置重试次数和重试频率；
2、生产端实现消息可靠性投递。

两种方法消费端都可能收到重复消息，要求消费端必须实现幂等性消费。



消息的可靠投递

生产端

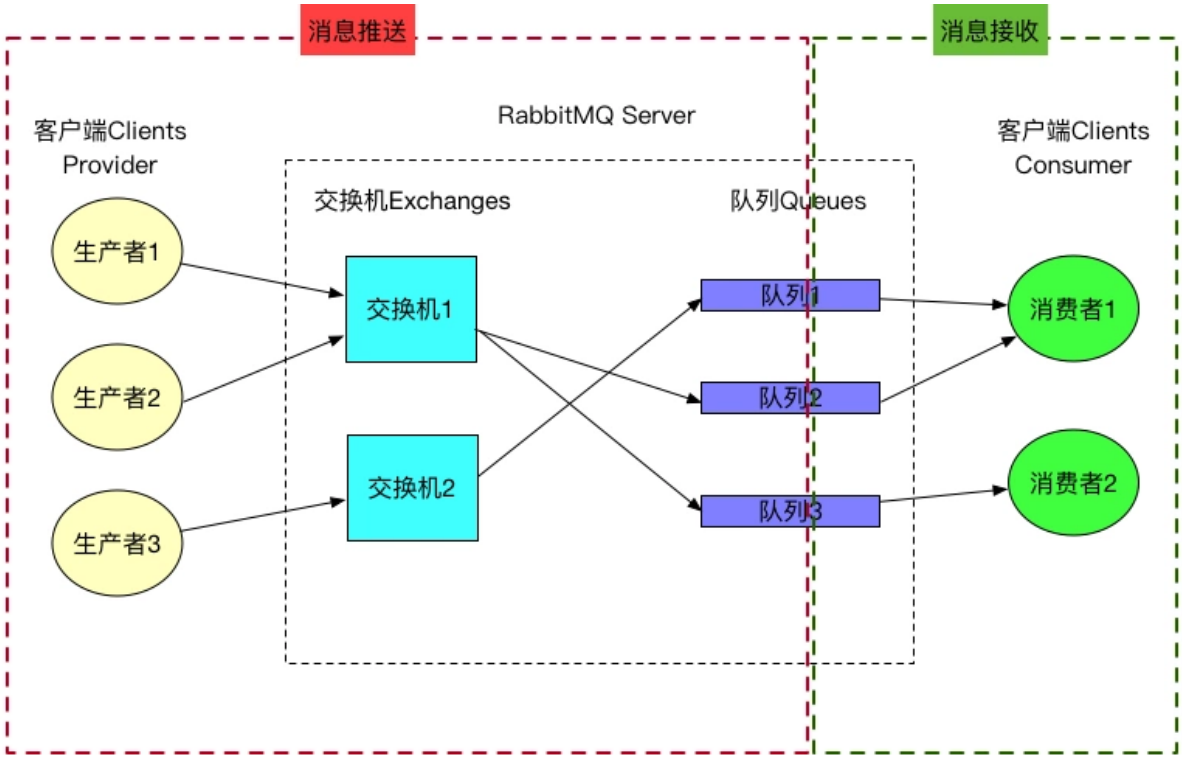
在使用 RabbitMQ 的时候，作为消息发送方希望杜绝任何消息丢失或者投递失败场景。RabbitMQ 为我们提供了两种方式用来控制消息的投递可靠性模式

- confirm 确认模式
- return 退回模式

消息投递到exchange的确认模式

rabbitmq的消息投递的过程为：

producer ——> rabbitmq broker cluster ——> exchange ——> queue ——> consumer



- 生产端发送消息到rabbitmq broker cluster后，异步接受从rabbitmq返回的ack确认信息
- 生产端收到返回的ack确认消息后，根据ack是true还是false，调用confirmCallback接口进行处理

1、**改yml**

```yaml
spring:
  #rabbitmq 连接配置
  rabbitmq:
    publisher-confirm-type: correlated # 开启confirm确认模式
```

2、**实现confirm方法**

实现ConfirmCallback接口中的confirm方法，消息只要被 rabbitmq broker接收到就会触ConfirmCallback 回调，ack为true表示消息发送成功，ack为false表示消息发送失败

```java
package com.rabbitmq.config;

import org.springframework.amqp.rabbit.connection.CorrelationData;
```

```java
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.stereotype.Component;

/**
 * 实现ConfirmCallback接口
 */
   @Component
public class ConfirmCallbackService implements
RabbitTemplate.ConfirmCallback {

    /**
     * @param correlationData 相关配置信息
     * @param ack exchange交换机  是否成功收到了消息。true 成功，false代表失败
     * @param cause 失败原因
     */
     @Override
     public void confirm(CorrelationData correlationData, boolean ack,
String cause) {
        if (ack) {
            //接收成功
            System.out.println("成功发送到交换机<===>");
        } else {
            //接收失败
            System.out.println("失败原因:===>" + cause);

            //TODO 做一些处理:消息再次发送等等

        }
     }
}
```

3、**测试**

定义 Exchange 和 Queue

定义交换机 confirmTestExchange 和队列 confirm_test_queue， 并将队列绑定在交换机上。

```java
package com.rabbitmq.config;

import org.springframework.amqp.core.Binding;
import org.springframework.amqp.core.BindingBuilder;
import org.springframework.amqp.core.FanoutExchange;
import org.springframework.amqp.core.Queue;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

/**
 * 队列与交换机绑定
 */
@Configuration
public class QueueConfig {

    @Bean(name = "confirmTestQueue")
    public Queue confirmTestQueue() {
        return new Queue("confirm_test_queue", true, false, false);
    }

    @Bean(name = "confirmTestExchange")
```

```
    public FanoutExchange confirmTestExchange() {
        return new FanoutExchange("confirmTestExchange");
    }

    @Bean
    public Binding confirmTestFanoutExchangeAndQueue(
            @Qualifier("confirmTestExchange") FanoutExchange confirmTestExchange,
            @Qualifier("confirmTestQueue") Queue confirmTestQueue) {
        return BindingBuilder.bind(confirmTestQueue).to(confirmTestExchange);
    }
}
```

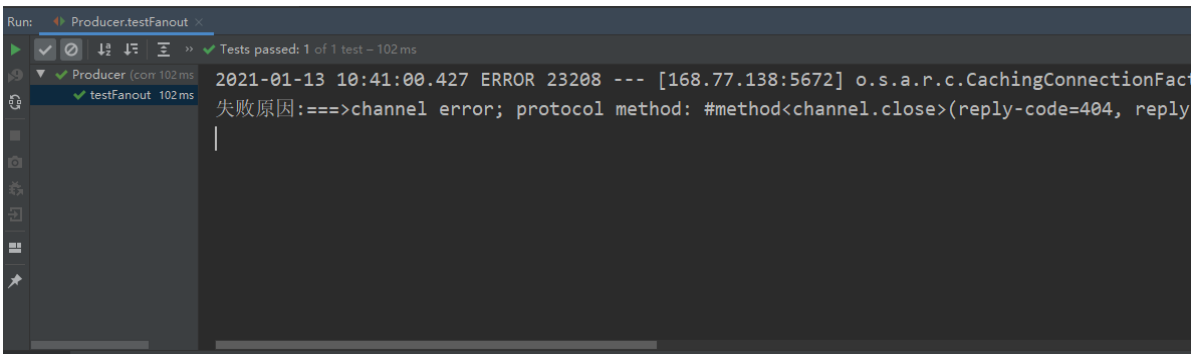
生产者

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes = RabbitmqApplication.class)
public class Producer {

    @Autowired
    private RabbitTemplate rabbitTemplate;  //注入rabbitmq对象

    @Autowired
    private ConfirmCallbackService confirmCallbackService; //注入ConfirmCallback对象

    @Test
    public void test() {
        //
        rabbitTemplate.setConfirmCallback(confirmCallbackService);
        //发送消息
        rabbitTemplate.convertAndSend("confirmTestExchange1", "", "hello,ConfirmCallback你好");
    }
}
```

正确情况，ack返回true，表示投递成功。

现在我们改变交换机名字，发送到一个不存在的交换机

```
//发送消息
rabbitTemplate.convertAndSend("confirmTestExchange1", "", "hello,ConfirmCallback你好");
```



消息未投递到queue的退回模式
消息从 exchange->queue 投递失败则会返回一个 returnCallback

生产端通过实现ReturnCallback接口，启动消息失败返回，消息路由不到队列时会触发该回调接口

1、**改yml**

```yaml
spring:
  # rabbitmq 连接配置
  rabbitmq:
    publisher-returns: true # 开启退回模式
```

2、**设置投递失败的模式**
如果消息没有路由到Queue，则丢弃消息（默认）

如果消息没有路由到Queue，返回给消息发送方ReturnCallBack（开启后）

```
rabbitTemplate.setMandatory(true);
```

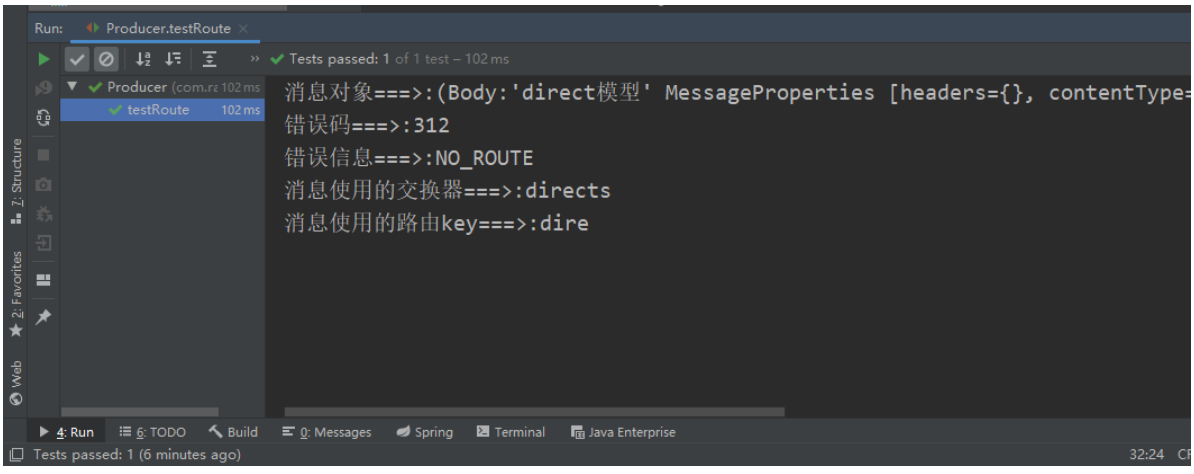
3、**实现returnedMessage方法**
启动消息失败返回，消息路由不到队列时会触发该回调接口

```
package com.rabbitmq.config;

import org.springframework.amqp.core.Message;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.stereotype.Component;

@Component
public class ReturnCallbackService implements RabbitTemplate.ReturnCallback
{

    /**
     * @param message    消息对象
     * @param replyCode  错误码
     * @param replyText  错误信息
     * @param exchange   交换机
     * @param routingKey 路由键
     */
     @Override
     public void returnedMessage(Message message, int replyCode, String
replyText, String exchange, String routingKey) {

        System.out.println("消息对象===>:" + message);
        System.out.println("错误码===>:" + replyCode);
        System.out.println("错误信息===>:" + replyText);
        System.out.println("消息使用的交换器===>:" + exchange);
        System.out.println("消息使用的路由key===>:" + routingKey);

        //TODO ===>做业务处理
     }
}
```

4、**测试**
生产者

```java
@RunWith(SpringRunner.class)
@SpringBootTest(classes = RabbitmqApplication.class)
public class Producer {

    @Autowired
    private RabbitTemplate rabbitTemplate;  //注入rabbitmq对象
    @Autowired
    private ConfirmCallbackService confirmCallbackService;
    @Autowired
    private ReturnCallbackService returnCallbackService;

    @Test
    public void test() {

        /**
         * 确保消息发送失败后可以重新返回到队列中
         */
        rabbitTemplate.setMandatory(true);

        /**
         * 消息投递到队列失败回调处理
         */
        rabbitTemplate.setReturnCallback(returnCallbackService);

        /**
         * 消息投递确认模式
         */
        rabbitTemplate.setConfirmCallback(confirmCallbackService);

        //发送消息
        rabbitTemplate.convertAndSend("confirmTestExchange", "info",
"hello,ConfirmCallback你好");
    }
}
```

如果不存在路由key"dire"，会调用ReturnCallback接口



消费端

消息确认机制ack
ack指Acknowledge确认。 表示消费端收到消息后的确认方式

消费端消息的确认分为：自动确认（默认）、手动确认、不确认

- AcknowledgeMode.NONE：不确认
- AcknowledgeMode.AUTO：自动确认
- AcknowledgeMode.MANUAL：手动确认

其中自动确认是指，当消息一旦被Consumer接收到，则自动确认收到，并将相应 message 从 RabbitMQ 的消息 缓存中移除。

但是在实际业务处理中，很可能消息接收到，业务处理出现异常，那么该消息就会丢失。如果设置了 手动确认方式，则需要在业务处理成功后，调用channel.basicAck()，手动签收，如果出现异常，则调用channel.basicNack()方法，让其自动重新发送消息。

1、**改yml**

```yaml
spring:
  rabbitmq:
    listener:
      simple:
        acknowledge-mode: manual # 手动确认
```

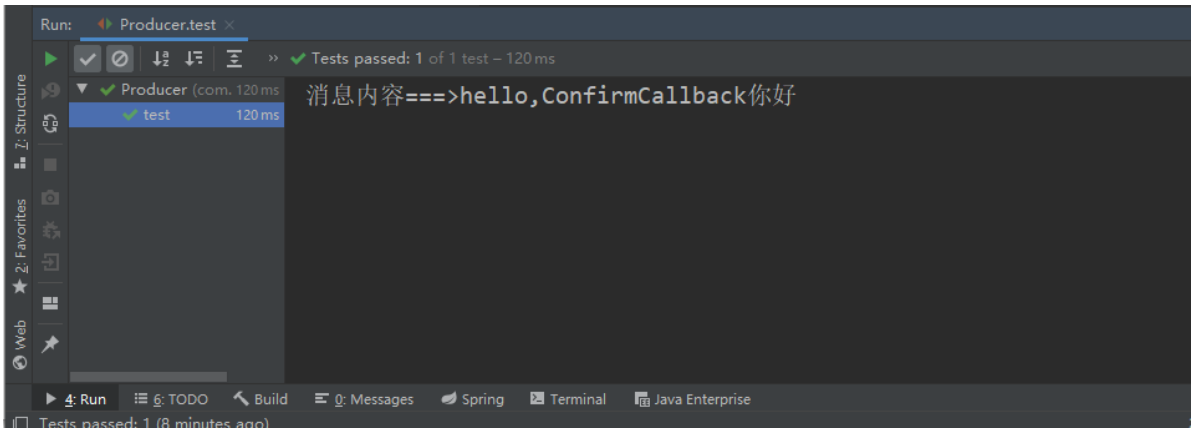
2、**确认配置**

```java
@Component
@RabbitListener(queues = "confirm_test_queue")
public class ReceiverMessage {

    @RabbitHandler
    public void processHandler(String msg, Channel channel, Message message) throws IOException {

        long deliveryTag = message.getMessageProperties().getDeliveryTag();
        try {

            System.out.println("消息内容===>" + new String(message.getBody()));

            //TODO 具体业务逻辑

            //手动签收[参数1:消息投递序号,参数2:批量签收]
            channel.basicAck(deliveryTag, true);
        } catch (Exception e) {
            //拒绝签收[参数1:消息投递序号,参数2:批量拒绝,参数3:是否重新加入队列]
            channel.basicNack(deliveryTag, true, true);
        }
    }
}
```

channel.basicNack 方法与 channel.basicReject 方法区别在于basicNack可以批量拒绝多条消息，而basicReject一次只能拒绝一条消息。

3、**测试**

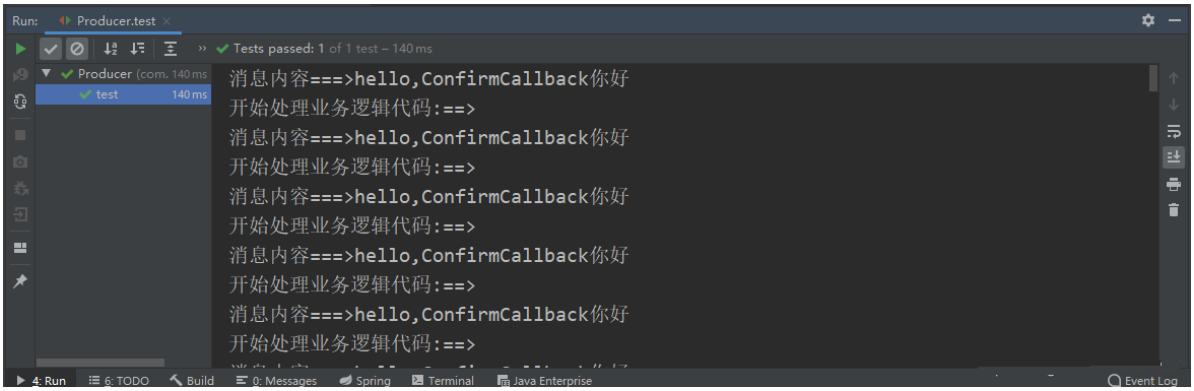
正常情况



异常情况

在业务处理模块增加异常

```
1  //TODO 具体业务逻辑
2  System.out.println("开始处理业务逻辑代码:==>");
3  int i = 3/0;
```

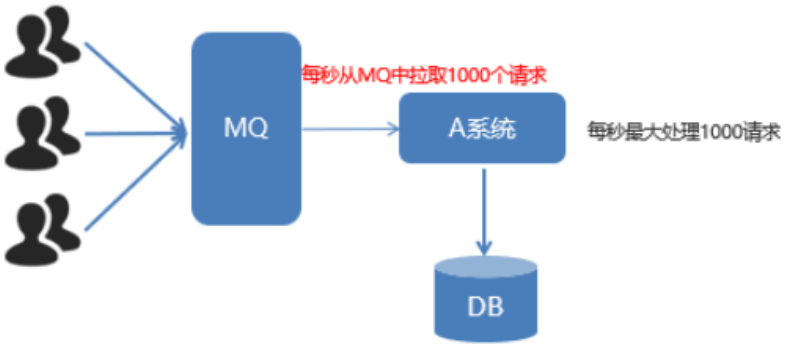


发生异常，拒绝确认，重新加入队列，一直循环，知道确认消息。

消费端限流

假设一个场景，首先，我们 Rabbitmq 服务器积压了有上万条未处理的消息，我们随便打开一个消费者客户端，会出现这样情况: 巨量的消息瞬间全部推送过来，但是我们单个客户端无法同时处理这么多数据!

当数据量特别大的时候，我们对生产端限流肯定是不科学的，因为有时候并发量就是特别大，有时候并发量又特别少，我们无法约束生产端，这是用户的行为。所以我们应该对消费端限流，用于保持消费端的稳定，当消息数量激增的时候很有可能造成资源耗尽，以及影响服务的性能，导致系统的卡顿甚至直接崩溃。



TTL

Time To Live，消息过期时间设置

声明队列时，指定即可

TTL:过期时间

1. 队列统一过期
2. 消息单独过期

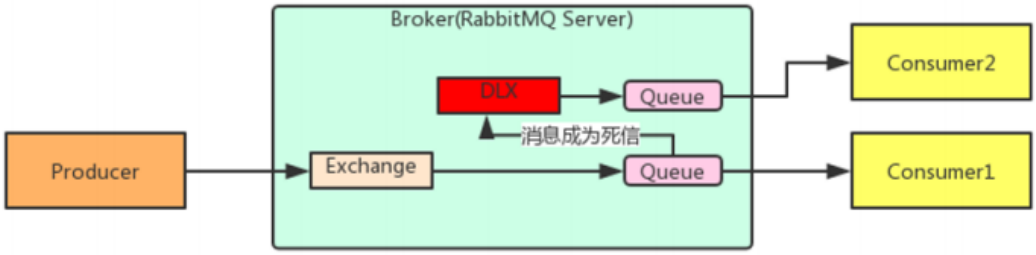
如果设置了消息的过期时间，也设置了队列的过期时间，它以时间短的为准。

- 队列过期后，会将队列所有消息全部移除
- 消息过期后，只有消息在队列顶端，才会判断其是否过期(移除掉)

死信队列

死信队列，英文缩写：DLX 。Dead Letter Exchange（死信交换机），当消息成为Dead message后，可以

被重新发送到另一个交换机，这个交换机就是DLX



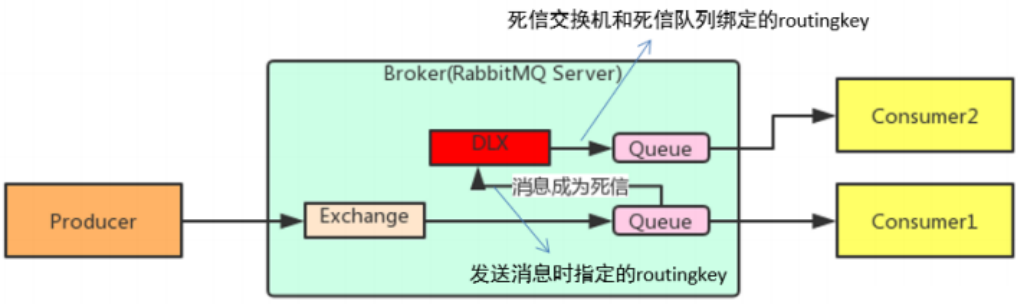
消息成为死信的三种情况：

1. 队列消息长度到达限制；
2. 消费者拒接消费消息，basicNack/basicReject,并且不把消息重新放入原目标队列,requeue=false；
3. 原队列存在消息过期设置，消息到达超时时间未被消费；

队列绑定死信交换机：

给队列设置参数： x-dead-letter-exchange 和 x-dead-letter-routing-key

也就是说此时Queue作为"生产者"



延迟队列

延迟队列，即消息进入队列后不会立即被消费，只有到达指定时间后，才会被消费

需求：

下单后，30分钟未支付，取消订单，回滚库存。

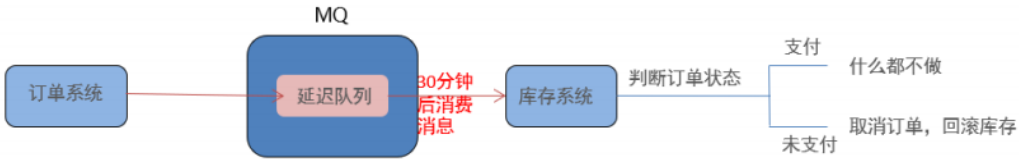
新用户注册成功7天后，发送短信问候。

实现方式：

定时器 （×）

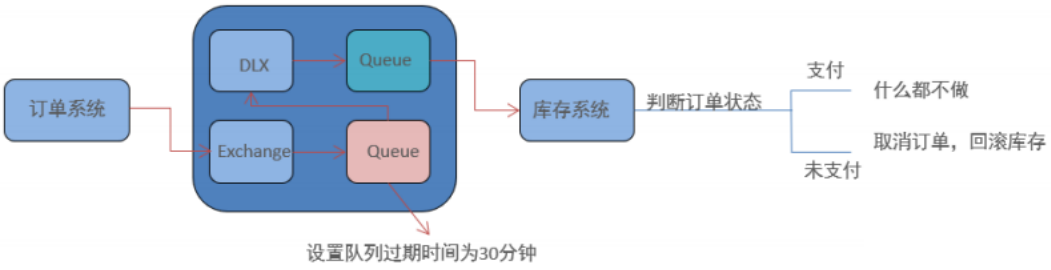
延迟队列 （√）

实现步骤：



在RabbitMQ中并未提供延迟队列功能

替代实现： TTL+死信队列 组合实现延迟队列的效果



设置队列过期时间30分钟，当30分钟过后，消息未被消费，进入死信队列，路由到指定队列，调用库存系统，判断订单状态。

RabbitMQ 监控

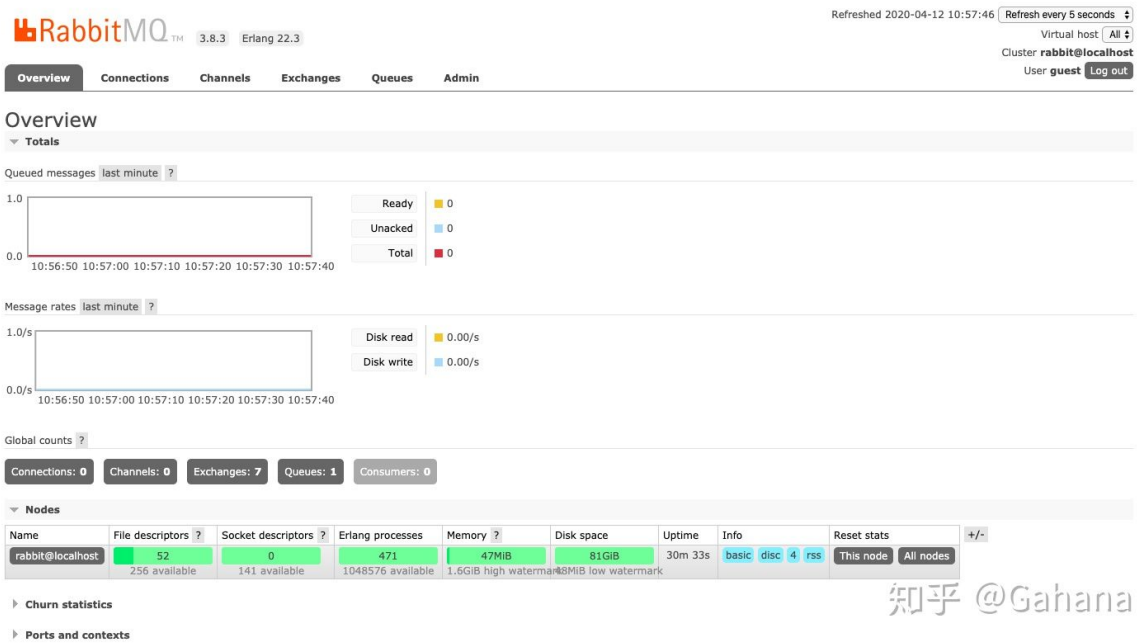
接下来说说监控的相关内容。

监控还是非常重要的，特别是在生产环境。磁盘满了，队列积压严重，如果我们还不知道，老板肯定会怀疑，莫不是这家伙要跑路？

而且我现在就遇到了这样的情况，主要是队列积压的问题。由于量不是很大，所以磁盘空间倒不是很担心，但有时程序执行会报错，导致队列一直消费不下去，这就很让人尴尬了。

查了一些资料，总结了一下。想要了解 RabbitMQ 的运行状态，主要有三种途径：Management UI，rabbitmqctl 命令和 REST API 以及使用 prometheus + grafana，当然大厂的话都会自定开发基于 api 监控系统。

Management UI



RabbitMQ 给我们提供了丰富的 Web 管理功能，通过页面，我们能看到 RabbitMQ 的整体运行状况，交换机和队列的状态等，还可以进行人员管理和权限配置，相当全面。

但如果想通过页面来监控，那出不出问题只能靠缘分。看到出问题了，是运气好，看不到出问题，那是必然。

这也是我当前的现状，所以为了避免出现大问题，得赶紧改变一下。

备注：通过 http://127.0.0.1:15672 来访问 Web 页面，默认情况下用户名和密码都是 guest，但生产环境下都应该改掉的。

rabbitmqctl 命令

与前端页面对应的就是后端的命令行命令了，同样非常丰富。平时自己测试，或者临时查看一些状态时，也能用得上。但就我个人使用感觉来说，用的并不是很多。

我总结一些还算常用的，列在下面，大家各取所需：

```
# 启动服务
rabbitmq-server

# 停止服务
rabbitmqctl stop

# vhost 增删查
rabbitmqctl add_vhost
rabbitmqctl delete_vhost
rabbitmqctl list_vhosts

# 查询交换机
rabbitmqctl list_exchanges

# 查询队列
rabbitmqctl list_queues

# 查看消费者信息
rabbitmqctl list_consumers

# user 增删查
rabbitmqctl add_user
rabbitmqctl delete_user
rabbitmqctl list_users
```

REST API

终于来到重点了，对于程序员来说，看到有现成的 API 可以调用，那真是太幸福了。

自动化监控和一些需要批量的操作，通过调用 API 来实现是最好的方式。比如有一些需要初始化的用户和权限，就可以通过脚本来一键完成，而不是通过页面逐个添加，简单又快捷。

下面是一些常用的 API：

```
# 概括信息
curl -i -u guest:guest http://localhost:15672/api/overview
```

```
# vhost 列表
curl -i -u guest:guest http://localhost:15672/api/vhosts

# channel 列表
curl -i -u guest:guest http://localhost:15672/api/channels

# 节点信息
curl -i -u guest:guest http://localhost:15672/api/nodes

# 交换机信息
curl -i -u guest:guest http://localhost:15672/api/exchanges

# 队列信息
curl -i -u guest:guest http://localhost:15672/api/queues
```

就我现在遇到的情况来说，`overview` 和 `queues` 这两个 API 就可以满足我的需求，大家也可以根据自己项目的实际情况来选择。

API 返回内容是 json，而且字段还是挺多的，刚开始看会感觉一脸懵，具体含义对照官网的解释和实际情况来慢慢琢磨，弄懂也不是很困难。

下面是一个demo代码，主要使用HttpClient以及jackson来调用相关参数。
相关maven如下：

```
<dependency>
    <groupId>org.apache.httpcomponents</groupId>
    <artifactId>httpclient</artifactId>
    <version>4.3.6</version>
</dependency>
<dependency>
 <groupId>com.fasterxml.jackson.core</groupId>
 <artifactId>jackson-databind</artifactId>
 <version>2.7.4</version>
</dependency>
<dependency>
 <groupId>com.fasterxml.jackson.core</groupId>
 <artifactId>jackson-annotations</artifactId>
 <version>2.7.4</version>
</dependency>
<dependency>
 <groupId>com.fasterxml.jackson.core</groupId>
 <artifactId>jackson-core</artifactId>
 <version>2.7.4</version>
</dependency>
```

```
impor
```

相关代码（有点长）：

```
package com.vms.test.zzh.rabbitmq.monitor;

import com.fasterxml.jackson.databind.DeserializationFeature;
import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.SerializationFeature;
```

```
7
8  import org.apache.http.HttpEntity;
9  import org.apache.http.auth.UsernamePasswordCredentials;
10 import org.apache.http.client.methods.CloseableHttpResponse;
11 import org.apache.http.client.methods.HttpGet;
12 import org.apache.http.impl.auth.BasicScheme;
13 import org.apache.http.impl.client.CloseableHttpClient;
14 import org.apache.http.impl.client.HttpClients;
15 import org.apache.http.util.EntityUtils;
16
17 import java.io.IOException;
18 import java.util.HashMap;
19 import java.util.Iterator;
20 import java.util.Map;
21
22 public class MonitorDemo {
23     public static void main(String[] args) {
24         try {
25             Map<String, ClusterStatus> map =
fetchNodesStatusData("http://localhsot:15672/api/nodes", "root", "root");
26             for (Map.Entry entry : map.entrySet()) {
27                 System.out.println(entry.getKey() + " : " +
entry.getValue());
28             }
29         } catch (IOException e) {
30             e.printStackTrace();
31         }
32     }
33
34     public static Map<String,ClusterStatus> fetchNodesStatusData(String
url, String username, String password) throws IOException {
35         Map<String, ClusterStatus> clusterStatusMap = new HashMap<String,
ClusterStatus>();
36         String nodeData = getData(url, username, password);
37         JsonNode jsonNode = null;
38         try {
39             jsonNode = JsonUtil.toJsonNode(nodeData);
40         } catch (IOException e) {
41             e.printStackTrace();
42         }
43         Iterator<JsonNode> iterator = jsonNode.iterator();
44         while (iterator.hasNext()) {
45             JsonNode next = iterator.next();
46             ClusterStatus status = new ClusterStatus();
47             status.setDiskFree(next.get("disk_free").asLong());
48             status.setFdUsed(next.get("fd_used").asLong());
49             status.setMemoryUsed(next.get("mem_used").asLong());
50             status.setProcUsed(next.get("proc_used").asLong());
51             status.setSocketUsed(next.get("sockets_used").asLong());
52             clusterStatusMap.put(next.get("name").asText(), status);
53         }
54         return clusterStatusMap;
55     }
56
57     public static String getData(String url, String username, String
password) throws IOException {
58         CloseableHttpClient httpClient = HttpClients.createDefault();
```

```
59          UsernamePasswordCredentials creds = new
UsernamePasswordCredentials(username, password);
60          HttpGet httpGet = new HttpGet(url);
61          httpGet.addHeader(BasicScheme.authenticate(creds, "UTF-8", false));
62          httpGet.setHeader("Content-Type", "application/json");
63          CloseableHttpResponse response = httpClient.execute(httpGet);
64
65          try {
66              if (response.getStatusLine().getStatusCode() != 200) {
67                  System.out.println("call http api to get rabbitmq data
return code: " + response.getStatusLine().getStatusCode() + ", url: " +
url);
68              }
69              HttpEntity entity = response.getEntity();
70              if (entity != null) {
71                  return EntityUtils.toString(entity);
72              }
73          } finally {
74              response.close();
75          }
76
77          return "";
78      }
79
80      public static class JsonUtil {
81          private static ObjectMapper objectMapper = new ObjectMapper();
82          static {
83
 objectMapper.disable(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES);
84              objectMapper.disable(SerializationFeature.FAIL_ON_EMPTY_BEANS);
85          }
86
87          public static JsonNode toJsonNode(String jsonString) throws
IOException {
88              return objectMapper.readTree(jsonString);
89          }
90      }
91
92      public static class ClusterStatus {
93          private long diskFree;
94          private long diskLimit;
95          private long fdUsed;
96          private long fdTotal;
97          private long socketUsed;
98          private long socketTotal;
99          private long memoryUsed;
100         private long memoryLimit;
101         private long procUsed;
102         private long procTotal;
103         // 此处省略了Getter和Setter方法
104         @Override
105         public String toString() {
106             return "ClusterStatus{" +
107                     "diskFree=" + diskFree +
108                     ", diskLimit=" + diskLimit +
109                     ", fdUsed=" + fdUsed +
110                     ", fdTotal=" + fdTotal +
111                     ", socketUsed=" + socketUsed +
```

```
                    ", socketTotal=" + socketTotal +
                    ", memoryUsed=" + memoryUsed +
                    ", memoryLimit=" + memoryLimit +
                    ", procUsed=" + procUsed +
                    ", procTotal=" + procTotal +
                    '}';
        }
    }
}
```

代码输出：

```
rabbit@zhuzhonghua2-fqawb : ClusterStatus{diskFree=34480971776, diskLimit=0, fdUsed=38, fdTotal=0, socketUsed=1, socketTotal=0, memoryUsed=44508400, memoryLimit=0, procUsed=196, procTotal=0}
rabbit@hiddenzhu-8drdc : ClusterStatus{diskFree=36540743680, diskLimit=0, fdUsed=28, fdTotal=0, socketUsed=1, socketTotal=0, memoryUsed=53331640, memoryLimit=0, procUsed=181, procTotal=0}
```

通过对返回结果进行解析，就可以判断 RabbitMQ 的整体运行状态，如果发生超阈值的情况，可以发送告警或邮件，来达到监控的效果。

针对队列积压情况的监控判断，有两种方式：

- 一是设置队列积压长度阈值，如果超过阈值即告警；
- 二是保存最近五次的积压长度，如果积压逐渐增长并超阈值，即告警。

第二种方式更好，判断更加精准，误告可能性小，但实现起来也更复杂。

这里只是提一个思路，等后续再把实践结果和代码分享出来。或者大家有哪些更好的方法吗？欢迎留言交流。

prometheus + grafana 监控rabbitmq

安装 docker-compose

前言

第一种：RabbitMQ内部集成Prometheus来获取指标

- 3.8.0之前版本，RabbitMQ可以使用单独的插件prometheus_rabbitmq_exporter来向Prometheus公开指标，要单独下载到RabbitMQ安装目录中进行安装；

  prometheus_rabbitmq_exporter：https://github.com/deadtrickster/prometheus_rabbitmq_exporter

- 3.8.0版开始，RabbitMQ附带了内置的Prometheus＆Grafana支持。虽然内置了该插件，但也要进行安装

  rabbitmq-prometheus：https://github.com/rabbitmq/rabbitmq-prometheus

第二种：使用独立程序来获取指标（RabbitMQ_exporter）

不管什么版本都能使用，要单独启动exporter进程

rabbitmq_exporter：https://github.com/kbudde/rabbitmq_exporter

RabbitMQ 官方监控介绍：

- https://www.rabbitmq.com/monitoring.html
- https://www.rabbitmq.com/prometheus.html#overview-prometheus

先介绍采用第二种方式实现。（因为企业中使用的rabbitmq基本都是3.8.x之前的版本居多。）

采用docker方式启动

一、使用独立程序来获取指标（RabbitMQ_exporter）

安装rabbitmq_exporter

注：在RabbitMQ集群下的任意一个节点部署它。

- **上传解压**

```
wget
https://github.com/kbudde/rabbitmq_exporter/releases/download/v0.29.0/rabbitmq_exporter-0.20.0.linux-amd64.tar.gz
tar -xvf rabbitmq_exporter-0.29.0.linux-amd64.tar
cd rabbitmq_exporter-0.29.0.linux-amd64/
```

- **配置**

使用默认配置

- **启动**

进入根目录下，输入以下命令：

```
cd /usr/local/rabbitmq_exporter-0.29.0.linux-amd64
RABBIT_USER=guest RABBIT_PASSWORD=guest OUTPUT_FORMAT=json PUBLIC_PORT=9090 RABBIT_URL=http://localhost:15672 nohup ./rabbitmq_exporter &
tail -1000f nohup.out
```

参数说明：

RABBIT_USER：rabbit用户名

RABBIT_PASSWORD：rabbit密码

RABBIT_URL：rabbit服务地址和端口

OUTPUT_FORMAT：输出格式

PUBLIC_PORT：暴露端口

启动成功后，可以访问 http://10.0.101.100:9090/metrics/ ，（IP和端口要改成相应环境的）

看抓取的信息如下：

```
# HELP go_gc_duration_seconds A summary of the GC invocation durations.
# TYPE go_gc_duration_seconds summary
go_gc_duration_seconds{quantile="0"} 8.413e-06
go_gc_duration_seconds{quantile="0.25"} 2.3593e-05
go_gc_duration_seconds{quantile="0.5"} 2.9758e-05
go_gc_duration_seconds{quantile="0.75"} 4.2374e-05
go_gc_duration_seconds{quantile="1"} 0.00029881
go_gc_duration_seconds_sum 2.589836342
go_gc_duration_seconds_count 50889
# HELP go_goroutines Number of goroutines that currently exist.
# TYPE go_goroutines gauge
go_goroutines 15
# HELP go_memstats_alloc_bytes Number of bytes allocated and still in use.
# TYPE go_memstats_alloc_bytes gauge
go_memstats_alloc_bytes 3.4816744e+07
```

使用docker-compose方式启动prometheus和grafana

下载老师上传的监控压缩包 rabbitmq-monitor.tar.gz，提取码：rppe

上传到服务器解压，参照上面的部署安装好docker和docker-compose，进入文件夹中执行：

```
docker-compose up -d

#查看服务启动是否正常
docker-compose ps
```

如果没用问题，就可以执行下面的步骤。

Prometheus配置

- **配置**

修改prometheus组件的prometheus.yml加入rabbitMQ节点：

vim prometheus.yml

```
# 采集rabbit exporter监控数据
- job_name: 'rabbit_exporter'
  scrape_interval: 60s
  scrape_timeout: 60s
  static_configs:
  - targets: ['10.[redacted].5:9090','10.[redacted].213:9090','10.[redacted].215:9090']
```

- **启动验证**

用以下命令重启它，然后查看targets：

```
docker-compose restart prometheus
```

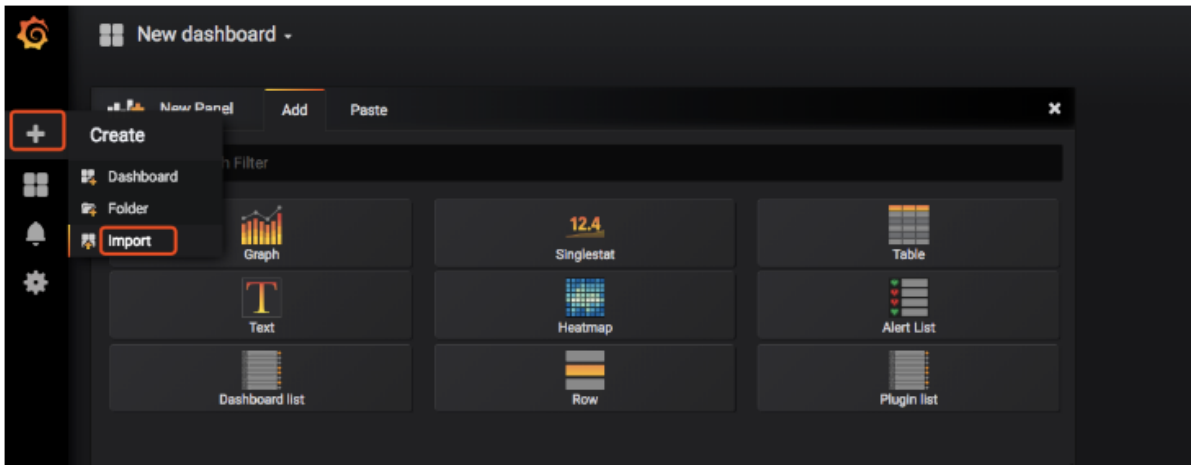
注：State=UP，说明成功

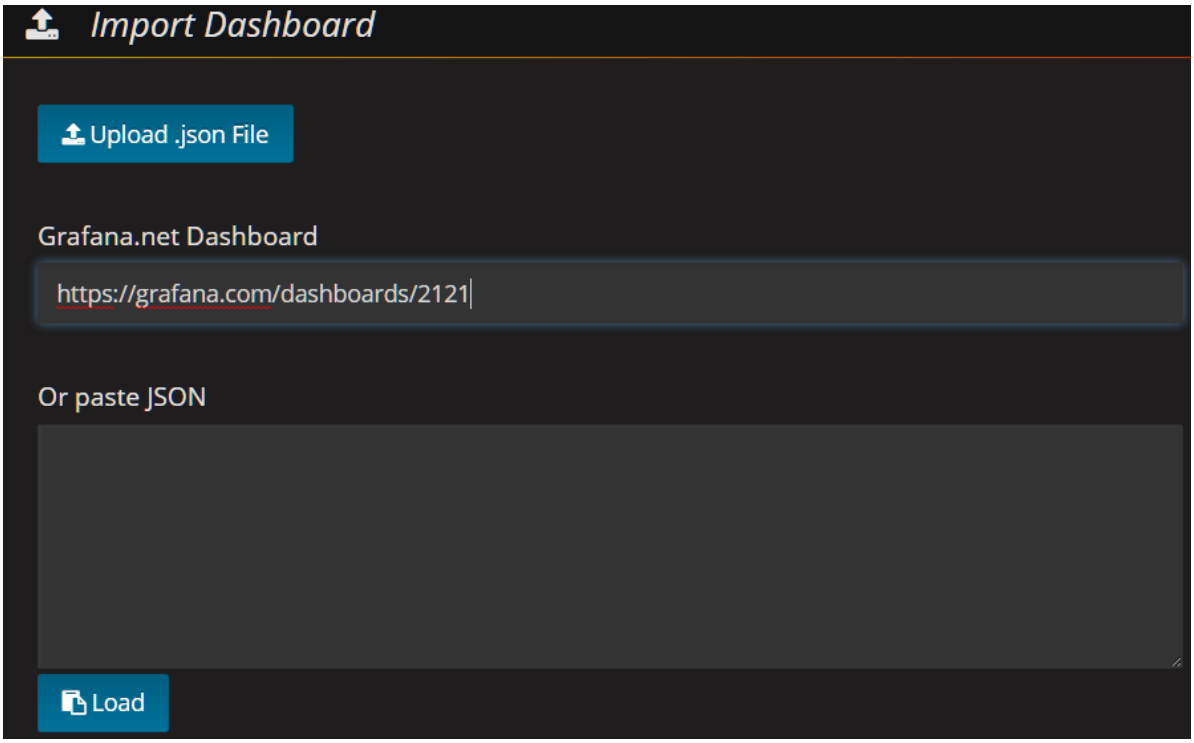
Grafana配置

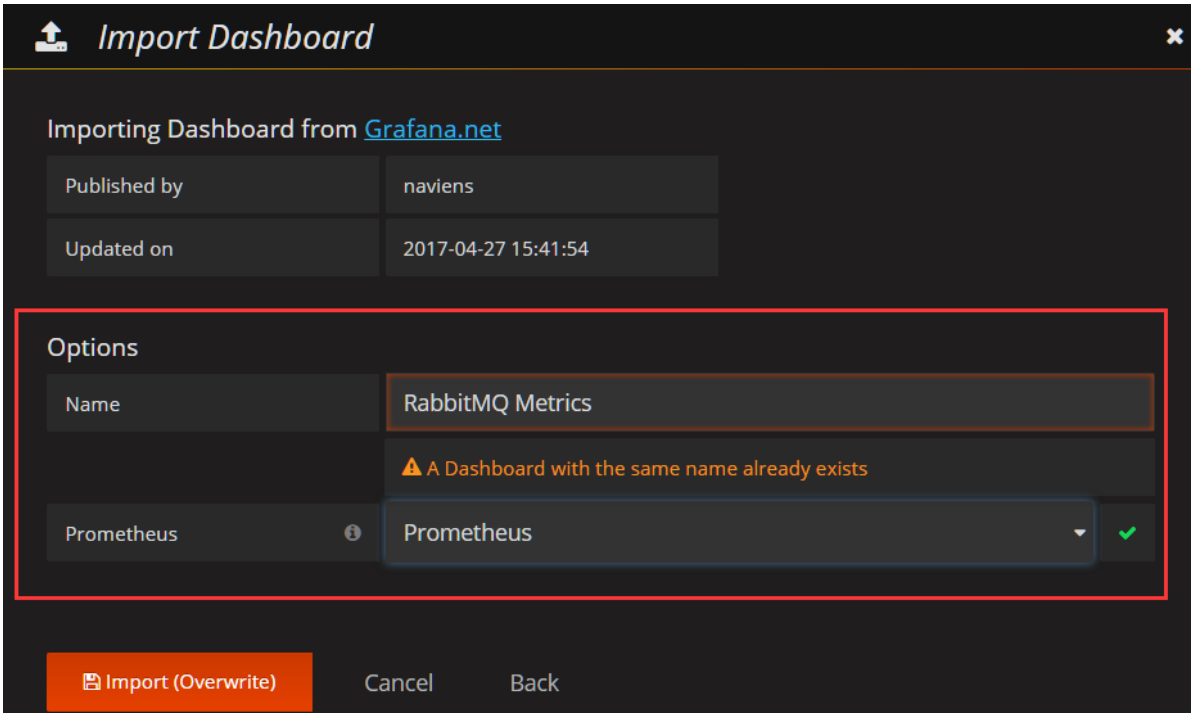
- **导入仪表盘模板**

通过浏览器访问：http://grafana服务器IP:3000

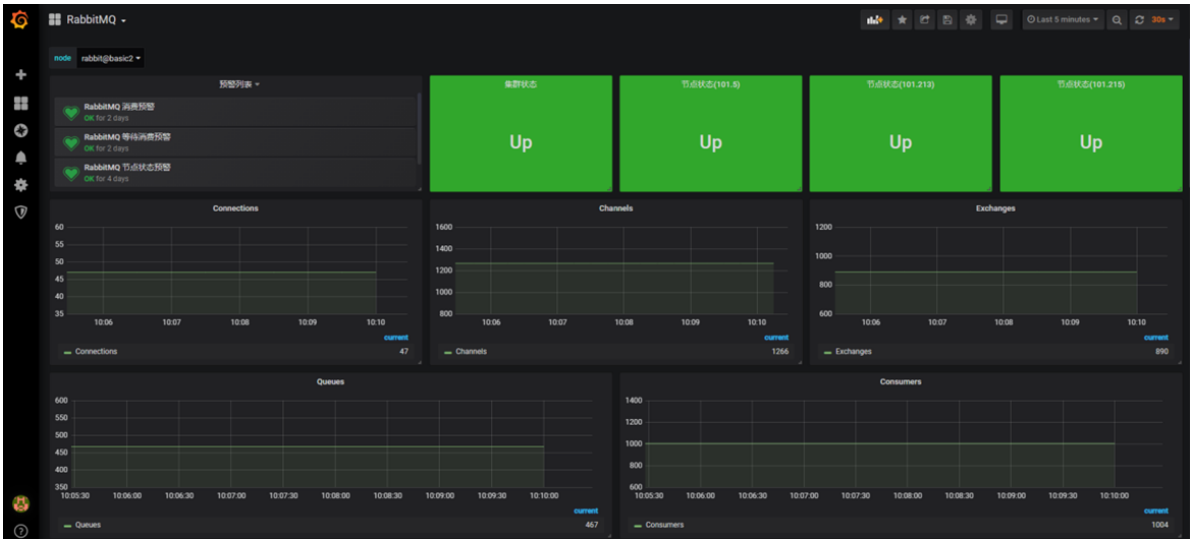
导入监控图表







以上仪表盘导入后再结合自身业务修改过的最终仪表盘：



- **预警指标**

序号	预警名称	预警规则	描述
1	集群状态预警	当集群状态不符合预期【!=1】时进行预警	
2	节点状态预警	当节点状态不符合预期【!=1】时进行预警	
3	等待消费预警	当等待消费的消息数量达到阈值【>1000】时进行预警	延迟消费
4	消费预警	当消费中的消息数量达到阈值【>1000】时进行预警	消费速度慢

二、RabbitMQ内部集成Prometheus来获取指标

由于官方文档已经介绍的非常详细，同时又docker-copose示例；所以这里就不再补充文档，大家请参考官方文档。

https://www.rabbitmq.com/prometheus.html