

RabbitMQ 实战教程

1.MQ引言

1.1 什么是MQ

MQ (Message Queue)：翻译为 消息队列，通过典型的 生产者 和 消费者 模型，生产者不断向消息队列中生产消息，消费者不断的从队列中获取消息。因为消息的生产和消费都是异步的，而且只关心消息的发送和接收，没有业务逻辑的侵入，轻松的实现系统间解耦。别名为 消息中间件 通过利用高效可靠的消息传递机制进行平台无关的数据交流，并基于数据通信来进行分布式系统的集成。

1.2 MQ有哪些

当今市面上有很多主流的消息中间件，如老牌的 ActiveMQ、RabbitMQ，炙手可热的 Kafka，阿里巴巴自主开发 RocketMQ 等。

1.3 不同MQ特点

```
1 # 1.ActiveMQ
2     ActiveMQ 是Apache出品，最流行的，能力强劲的开源消息总线。它是一个完全支持JMS
3     规范的消息中间件。丰富的API，多种集群架构模式让ActiveMQ在业界成为老牌的消息中间件，在中
4     小型企业颇受欢迎！
5
6 # 2.Kafka
7     Kafka是LinkedIn开源的分布式发布-订阅消息系统，目前归属于Apache顶级项目。
8     Kafka主要特点是基于Pull的模式来处理消息消费，
9     追求高吞吐量，一开始的目的就是用于日志收集和传输。0.8版本开始支持复制，不支持事
10    务，对消息的重复、丢失、错误没有严格要求，
11    适合产生大量数据的互联网服务的数据收集业务。
12
13
14 # 3.RocketMQ
15     RocketMQ是阿里开源的消息中间件，它是纯Java开发，具有高吞吐量、高可用性、适合
16     大规模分布式系统应用的特点。RocketMQ思路起
17     源于kafka，但并不是kafka的一个Copy，它对消息的可靠传输及事务性做了优化，目前
18     在阿里集团被广泛应用于交易、充值、流计算、消
          息推送、日志流式处理、binglog分发等场景。
# 4.RabbitMQ
        RabbitMQ是使用Erlang语言开发的开源消息队列系统，基于AMQP协议来实现。AMQP的主要特征是面向消息、队列、路由（包括点对点和
发布/订阅）、可靠性、安全。AMQP协议更多用在企业系统内对数据一致性、稳定性和可靠性要求很高的场景，对性能和吞吐量的要求还在
其次。
```

RabbitMQ比Kafka可靠，Kafka更适合IO高吞吐的处理，一般应用在大数据日志处理或对实时性（少量延迟），可靠性（少量丢数据）要求稍低的场景使用，比如ELK日志收集。

2.RabbitMQ 的引言

2.1 RabbitMQ

基于AMQP协议，erlang语言开发，是部署最广泛的开源消息中间件，是最受欢迎的开源消息中间件之一。



RabbitMQ is the most widely deployed open source message broker.

With tens of thousands of users, RabbitMQ is one of the most popular open source message brokers. From [T-Mobile](#) to [Runtastic](#), RabbitMQ is used worldwide at small startups and large enterprises.

Updates

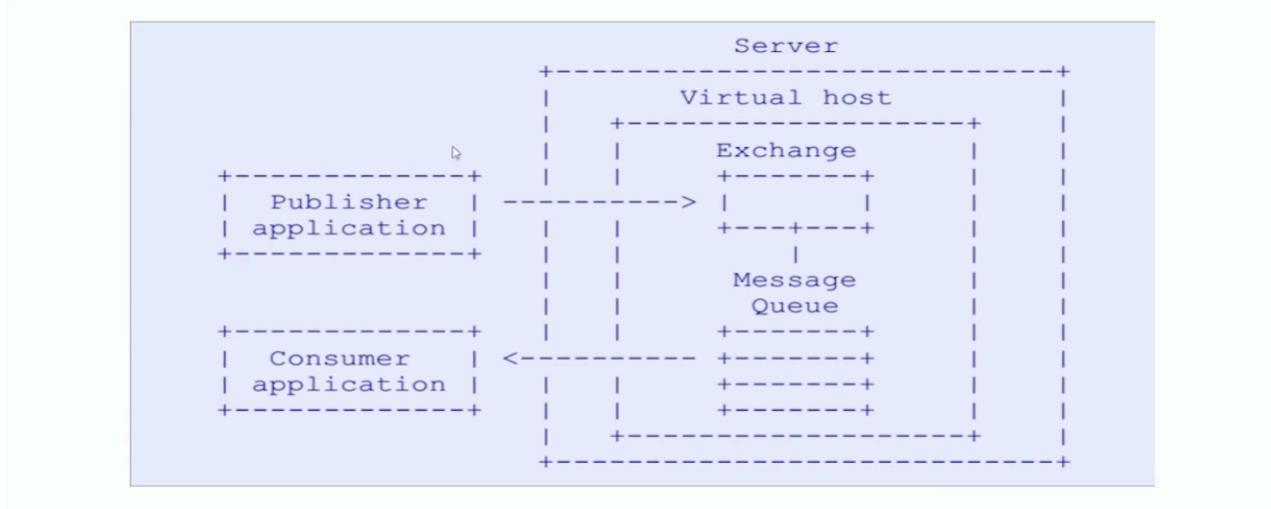
1. [RabbitMQ 3.7.18版本](#) 18 Sep 2019
2. [Erlang 20.3.x 支持退休时间表](#) 30 Jul 2019
3. [RabbitMQ 3.7.17版本](#) 29 Jul 2019

官网：<https://www.rabbitmq.com/>

官方教程：<https://www.rabbitmq.com/#getstarted>

1 # AMQP 协议

2 AMQP (advanced message queuing protocol) 在2003年时被提出，最早用于解决金融领不同平台之间的消息传递交互问题。顾名思义，AMQP是一种协议，更准确的说是一种binary wire-level protocol (链接协议)。这是其和JMS的本质差别，AMQP不从API层进行限定，而是直接定义网络交换的数据格式。这使得实现了AMQP的provider天然性就是跨平台的。以下是AMQP协议模型：



2.2 RabbitMQ 的安装

2.2.1 下载

官网下载地址：<https://www.rabbitmq.com/download.html>

Downloading and Installing RabbitMQ

The latest release of RabbitMQ is **3.7.18**. See [change log](#) for release notes. See [RabbitMQ support timeline](#) to find out what release series are supported.

RabbitMQ Server

Installation Guides

- Linux, BSD, UNIX: [Debian](#), [Ubuntu](#), [RHEL](#), [CentOS](#), [Fedora](#), [Generic binary build](#), [Solaris](#)
- Windows: [Installer \(recommended\)](#), [Binary build](#)
- MacOS: [Homebrew](#), [Generic binary build](#)
- [Erlang/OTP for RabbitMQ](#)

centos

window

mac

最新版本：3.7.18

2.2.2 下载的安装包



注意：这里的安装包是centos7安装的包

2.2.3 安装步骤

```

1 # 1. 将rabbitmq安装包上传到linux系统中
2      erlang-22.0.7-1.el7.x86_64.rpm
3      rabbitmq-server-3.7.18-1.el7.noarch.rpm
4
5 # 2. 安装Erlang依赖包
6      rpm -ivh erlang-22.0.7-1.el7.x86_64.rpm

```

```

7
8 # 3.安装RabbitMQ安装包(需要联网)
9     yum install -y rabbitmq-server-3.7.18-1.el7.noarch.rpm
10    注意:默认安装完成后配置文件模板在:/usr/share/doc/rabbitmq-server-
11    3.7.18/rabbitmq.config.example目录中,需要
12    将配置文件复制到/etc/rabbitmq/目录中,并修改名称为rabbitmq.config
13 # 4.复制配置文件
14     cp /usr/share/doc/rabbitmq-server-3.7.18/rabbitmq.config.example
15     /etc/rabbitmq/rabbitmq.config
16
17
18 # 5.查看配置文件位置
19     ls /etc/rabbitmq/rabbitmq.config
20
21
22 # 6.修改配置文件(参见下图:)
23     vim /etc/rabbitmq/rabbitmq.config

```

```

55     %% The default "guest" user is only permitted to access the server
56     %% via a loopback interface (e.g. localhost).
57     %% {loopback_users, [<<"guest">>]},
58     %%
59     %% Uncomment the following line if you want to allow access to the
60     %% guest user from anywhere on the network.
61     %% [{loopback_users, []}],
62

```

将上图中配置文件中红色部分去掉`%%`,以及最后的`,`逗号 修改为下图:

```

58     %%
59     %% Uncomment the following line if you want to allow access to the
60     %% guest user from anywhere on the network.
61     [{loopback_users, []}]
62

```

```

1 # 7.执行如下命令,启动rabbitmq中的插件管理
2     rabbitmq-plugins enable rabbitmq_management
3
4     出现如下说明:
5         Enabling plugins on node rabbit@localhost:
6             rabbitmq_management
7             The following plugins have been configured:
8                 rabbitmq_management
9                 rabbitmq_management_agent
10                rabbitmq_web_dispatch
11                Applying plugin configuration to rabbit@localhost...
12                The following plugins have been enabled:
13                    rabbitmq_management
14                    rabbitmq_management_agent
15                    rabbitmq_web_dispatch
16
17                    set 3 plugins.
18                    Offline change; changes will take effect at broker restart.
19

```

```

20 # 8.启动RabbitMQ的服务
21     systemctl start rabbitmq-server
22     systemctl restart rabbitmq-server
23     systemctl stop rabbitmq-server
24
25
26 # 9.查看服务状态(见下图:)
27     systemctl status rabbitmq-server
28 ● rabbitmq-server.service - RabbitMQ broker
29   Loaded: loaded (/usr/lib/systemd/system/rabbitmq-server.service;
disabled; vendor preset: disabled)
30     Active: active (running) since 三 2019-09-25 22:26:35 CST; 7s ago
31       Main PID: 2904 (beam.smp)
32         Status: "Initialized"
33       CGroup: /system.slice/rabbitmq-server.service
34           ├─2904 /usr/lib64/erlang/erts-10.4.4/bin/beam.smp -W w -A 64
35 -MBas ageffcbf -MHas ageffcbf -
36             MBlmbcs...
37             ├─3220 erl_child_setup 32768
38             ├─3243 inet_gethost 4
39             └─3244 inet_gethost 4
40
41 .....
```

```

[root@localhost ~]# rabbitmq-plugins enable rabbitmq_management
Enabling plugins on node rabbit@localhost:
rabbitmq_management
The following plugins have been configured:
  rabbitmq_management
  rabbitmq_management_agent
  rabbitmq_web_dispatch
Applying plugin configuration to rabbit@localhost...
The following plugins have been enabled:
  rabbitmq_management
  rabbitmq_management_agent
  rabbitmq_web_dispatch

set 3 plugins.
Offline change; changes will take effect at broker restart.
[root@localhost ~]# systemctl start rabbitmq-server
[root@localhost ~]# systemctl status rabbitmq-server
● rabbitmq-server.service - RabbitMQ broker
  Loaded: loaded (/usr/lib/systemd/system/rabbitmq-server.service; disabled; vendor preset: disabled)
  Active: active (running) since 三 2019-09-25 22:26:35 CST; 7s ago
    Main PID: 2904 (beam.smp)
      Status: "Initialized"
    CGroup: /system.slice/rabbitmq-server.service
        ├─2904 /usr/lib64/erlang/erts-10.4.4/bin/beam.smp -W w -A 64 -MBas ageffcbf -MHas ageffcbf -MBlmbcs...
        ├─3220 erl_child_setup 32768
        ├─3243 inet_gethost 4
        └─3244 inet_gethost 4

9月 25 22:26:35 localhost.localdomain rabbitmq-server[2904]: ## ##
9月 25 22:26:35 localhost.localdomain rabbitmq-server[2904]: ## ##      RabbitMQ 3.7.18. Copyright (C) 20...nc.
9月 25 22:26:35 localhost.localdomain rabbitmq-server[2904]: ##### Licensed under the MPL. See http...om/
9月 25 22:26:35 localhost.localdomain rabbitmq-server[2904]: ##### ##
9月 25 22:26:35 localhost.localdomain rabbitmq-server[2904]: ##### Logs: /var/log/rabbitmq/rabbit@lo...log
9月 25 22:26:35 localhost.localdomain rabbitmq-server[2904]: /var/log/rabbitmq/rabbit@localhost_upgrade.log

```

```

1 # 10.关闭防火墙服务
2     systemctl disable firewalld
3     Removed symlink /etc/systemd/system/multi-
4     user.target.wants/firewalld.service.
5     Removed symlink /etc/systemd/system/dbus-
6     org.fedoraproject.FirewallD1.service.
7     systemctl stop firewalld
8
9 # 11.访问web管理界面
10 http://10.15.0.8:15672/

```



```

1 # 12.登录管理界面
2     username: guest
3     password: guest

```

Refreshed 2019-09-26 19:49:45 Refresh every 5 seconds

Virtual host All

Cluster rabbit@localhost User guest Log out

Overview

Totals

Queued messages last minute ?

Currently idle

Message rates last minute ?

Currently idle

Global counts ?

Connections: 0 Channels: 0 Exchanges: 7 Queues: 0 Consumers: 0

Nodes

Name	File descriptors ?	Socket descriptors ?	Erlang processes	Memory ?	Disk space	Uptime	Info	Reset stats	+/-
rabbit@localhost	34 32768 available	0 29399 available	390 1048576 available	70MiB 390MiB high watermark	15GiB 48MiB low watermark	45m 44s	basic disc 1 rss	This node All nodes	+/-

Churn statistics

Ports and contexts

Export definitions

Import definitions

HTTP API Server Docs Tutorials Community Support Community Slack Commercial Support Plugins GitHub Changelog

3. RabbitMQ 配置

3.1 RabbitMQ 管理命令行

```
1 # 1.服务启动相关
2     systemctl start|restart|stop|status rabbitmq-server
3
4 # 2.管理命令行 用来在不使用web管理界面情况下命令操作RabbitMQ
5     rabbitmqctl help 可以查看更多命令
6
7 # 3.插件管理命令行
8     rabbitmq-plugins enable|list|disable
```

3.2 web管理界面介绍

3.2.1 overview概览

The screenshot shows the RabbitMQ Management Overview page. At the top, it displays system statistics: Erlang 22.0.7, Node: rabbit@localhost, Uptime: 33m 19s, and Memory usage: 70% (404 MB used). Below this, the 'Nodes' section shows a single node with 35 file descriptors and 0 socket descriptors. The 'Ports and contexts' section lists listening ports: AMQP (5672), AMQP (25672), and HTTP (15672). Arrows point from these port numbers to labels 'AMQP 协议端口' and 'http 端口'. At the bottom, there are sections for 'Export definitions' and 'Import definitions'.

- **connections:** 无论生产者还是消费者，都需要与 RabbitMQ 建立连接后才可以完成消息的生产和消费，在这里可以查看连接情况。
- **channels:** 通道，建立连接后，会形成通道，消息的投递获取依赖通道。
- **Exchanges:** 交换机，用来实现消息的路由
- **Queues:** 队列，即消息队列，消息存放在队列中，等待消费，消费后被移除队列。

3.2.2 Admin用户和虚拟主机管理

1. 添加用户

上面的Tags选项，其实是指定用户的角色，可选的有以下几个：

- 超级管理员 (administrator)

可登陆管理控制台，可查看所有的信息，并且可以对用户，策略(policy)进行操作。

- 监控者 (monitoring)

可登陆管理控制台，同时可以查看rabbitmq节点的相关信息(进程数，内存使用情况，磁盘使用情况等)

- 策略制定者 (policymaker)

可登陆管理控制台，同时可以对policy进行管理。但无法查看节点的相关信息(上图红框标识的部分)。

- 普通管理者 (management)

仅可登陆管理控制台，无法看到节点信息，也无法对策略进行管理。

- 其他

无法登陆管理控制台，通常就是普通的生产者和消费者。

2. 创建虚拟主机

1 # 虚拟主机

为了让各个用户可以互不干扰的工作，RabbitMQ添加了虚拟主机 (virtual Hosts) 的概念。其实就是一个独立的访问路径，不同用户使用不同路径，各自有自己的队列、交换机，互相不会影响。

3. 绑定虚拟主机和用户

创建好虚拟主机，我们还要给用户添加访问权限：

点击添加好的虚拟主机：

Virtual Hosts

Add a new virtual host

Name: []

HTTP API Server Docs Tutorials Community Support Community Slack Commercial Support Plugins GitHub Changelog

进入虚拟机设置界面:

Virtual Host: /elasticsearch

Overview

Queued messages last minute: ?

Waiting for data: ..

Message rates last minute: ?

Current idle: Data rate last minute: Waiting for data: ..

Details

Tracing enabled: ○ State: rabbit@localhost: running

Permissions

User: chenyu

Configure regexp: Write regexp: Read regexp: []

Set permission: 1

Topic permissions

User: chenyu

Exchange: (AMQP default) .*

Write regexp: .*

Read regexp: .*

Set topic permission: 2

Set topic permission: 3

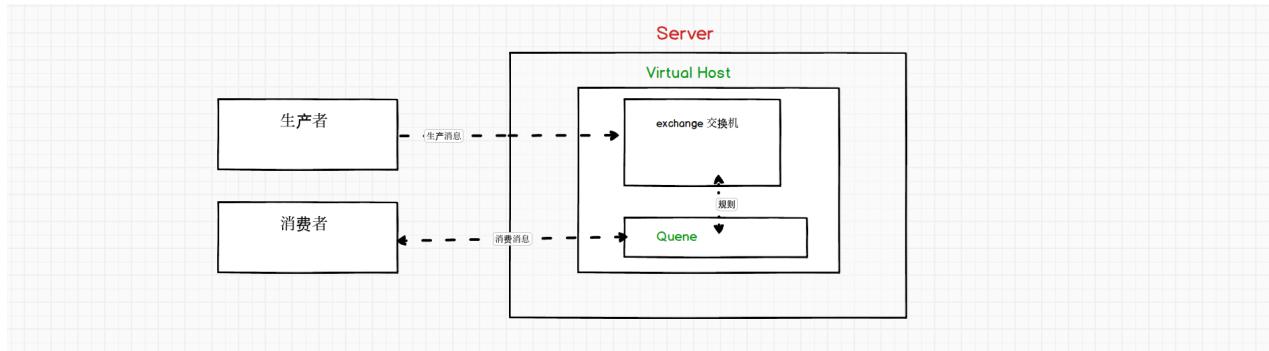
Set topic permission: 4

Delete this vhost

HTTP API Server Docs Tutorials Community Support Community Slack Commercial Support Plugins GitHub Changelog

4.RabbitMQ 的第一个程序

4.0 AMQP协议的回顾



4.1 RabbitMQ支持的消息模型

1 "Hello World!"

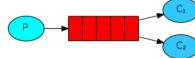
The simplest thing that does something



- [Python](#)
- [Java](#)
- [Ruby](#)
- [PHP](#)
- [C#](#)
- [JavaScript](#)
- [Go](#)
- [Elixir](#)
- [Objective-C](#)
- [Swift](#)
- [Spring AMQP](#)

2 Work queues

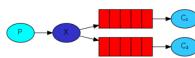
Distributing tasks among workers (the [competing consumers pattern](#))



- [Python](#)
- [Java](#)
- [Ruby](#)
- [PHP](#)
- [C#](#)
- [JavaScript](#)
- [Go](#)
- [Elixir](#)
- [Objective-C](#)
- [Swift](#)
- [Spring AMQP](#)

3 Publish/Subscribe

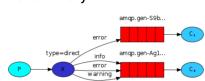
Sending messages to many consumers at once



- [Python](#)
- [Java](#)
- [Ruby](#)
- [PHP](#)
- [C#](#)
- [JavaScript](#)
- [Go](#)
- [Elixir](#)
- [Objective-C](#)
- [Swift](#)
- [Spring AMQP](#)

4 Routing

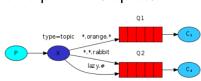
Receiving messages selectively



- [Python](#)
- [Java](#)
- [Ruby](#)
- [PHP](#)
- [C#](#)
- [JavaScript](#)
- [Go](#)
- [Elixir](#)
- [Objective-C](#)
- [Swift](#)
- [Spring AMQP](#)

5 Topics

Receiving messages based on a pattern (topics)



- [Python](#)
- [Java](#)
- [Ruby](#)
- [PHP](#)
- [C#](#)
- [JavaScript](#)
- [Go](#)
- [Elixir](#)
- [Objective-C](#)
- [Swift](#)
- [Spring AMQP](#)

6 RPC

[Request/reply.pattern](#)

example

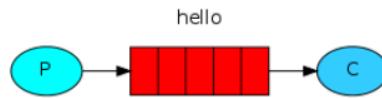


- [Python](#)
- [Java](#)
- [Ruby](#)
- [PHP](#)
- [C#](#)
- [JavaScript](#)
- [Go](#)
- [Elixir](#)
- [Objective-C](#)
- [Swift](#)
- [Spring AMQP](#)

4.2 引入依赖

```
1 <dependency>
2   <groupId>com.rabbitmq</groupId>
3   <artifactId>amqp-client</artifactId>
4   <version>5.7.2</version>
5 </dependency>
```

4.3 第一种模型(直连)



在上图的模型中，有以下概念：

- P: 生产者，也就是要发送消息的程序
- C: 消费者：消息的接受者，会一直等待消息到来。
- queue: 消息队列，图中红色部分。类似一个邮箱，可以缓存消息；生产者向其中投递消息，消费者从其中取出消息。

1. 开发生产者

```
1 //创建连接mq的连接工厂对象
2 ConnectionFactory connectionFactory = new ConnectionFactory();
3 //设置连接rabbitmq主机
4 connectionFactory.setHost("10.15.0.9");
5 //设置端口号
6 connectionFactory.setPort(5672);
7 //设置访问虚拟主机的用户名和密码
8 connectionFactory.setUsername("ems");
9 connectionFactory.setPassword("123");
10 //设置连接哪个虚拟主机
11 connectionFactory.setVirtualHost("/ems");
12 Connection connection = connectionFactory.newConnection();
13 //创建通道
14 Channel channel = connection.createChannel();
15 //通过通道绑定对应的消息队列
16 //参数1:队列名称 (如果队列不存在会自动创建)
17 //参数2: 用来定义数据是否持久化 true持久化队列 false不持久化队列
18 //参数3:是否独占队列 true独占队列 false不独占
19 //参数4:队列是否自动删除 (消息为空且没有被监听时) true自动删除队列 false不自动删除队列
20 //参数5:其他属性
21 channel.queueDeclare("hello",true,false,false,null);
22 //发布消息
23 //参数1:交换机名称 参数2:队列名称 参数3:传递消息的额外设置 参数4:消息的具体内容
24 //参数3:如果需要将消息队列中的数据持久化吗, 那么这里设置为
25     MessageProperties.PERSISTENT_TEXT_PLAIN
26     channel.basicPublish("", "hello", null, "hello rabbitmq".getBytes());
27     channel.close();
28     connection.close();
```

2. 开发消费者

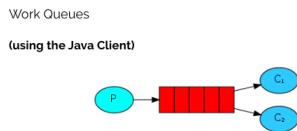
```
1 //创建连接工厂
2 ConnectionFactory connectionFactory = new ConnectionFactory();
3 connectionFactory.setHost("10.15.0.9");
4 connectionFactory.setPort(5672);
5 connectionFactory.setUsername("ems");
6 connectionFactory.setPassword("123");
7 connectionFactory.setVirtualHost("/ems");
8 //创建连接对象
9 Connection connection = connectionFactory.newConnection();
10 //创建通道
11 Channel channel = connection.createChannel();
12 //通道绑定对象
13 channel.queueDeclare("hello", true, false, false, null);
14 //消费消息
15 //参数1：消费哪个队列的消息，即队列名称
16 //参数2：开启消息的自动确认机制，消费者自动向rabbitmq确认消息消费
17 //参数3：消费时的回调接口
18 channel.basicConsume("hello",true,new DefaultConsumer(channel){
19     //最后一个参数：消息队列中取出的消息
20     @Override
21     public void handleDelivery(String consumerTag, Envelope envelope,
22         AMQP.BasicProperties properties, byte[] body) throws IOException {
23         System.out.println(new String(body));
24     }
25 });
26 );
```

3. 参数的说明

```
1 channel.queueDeclare("hello",true,false,false,null);
2 '参数1'：用来声明通道对应的队列
3 '参数2'：用来指定是否持久化队列
4 '参数3'：用来指定是否独占队列
5 '参数4'：用来指定是否自动删除队列
6 '参数5'：对队列的额外配置
```

4.4 第二种模型(work queue)

Work queues，也被称为（Task queues），任务模型。当消息处理比较耗时的时候，可能生产消息的速度会远远大于消息的消费速度。长此以往，消息就会堆积越来越多，无法及时处理。此时就可以使用work 模型：让多个消费者绑定到一个队列，共同消费队列中的消息。队列中的消息一旦消费，就会消失，因此任务是不会被重复执行的。



角色：

- P: 生产者：任务的发布者
- C1: 消费者-1，领取任务并且完成任务，假设完成速度较慢
- C2: 消费者-2：领取任务并完成任务，假设完成速度快

1. 开发生产者

```

1 channel.queueDeclare("hello", true, false, false, null);
2 for (int i = 0; i < 10; i++) {
3     channel.basicPublish("", "hello", null, (i+"====>:我是消息").getBytes());
4 }
```

2. 开发消费者-1

```

1 channel.queueDeclare("hello",true,false,false,null);
2 channel.basicConsume("hello",true,new DefaultConsumer(channel){
3     @Override
4     public void handleDelivery(String consumerTag, Envelope envelope,
5         AMQP.BasicProperties properties, byte[] body) throws IOException {
6         System.out.println("消费者1: "+new String(body));
7     }
});
```

3. 开发消费者-2

```

1 channel.queueDeclare("hello",true,false,false,null);
2 channel.basicConsume("hello",true,new DefaultConsumer(channel){
3     @Override
4     public void handleDelivery(String consumerTag, Envelope envelope,
5         AMQP.BasicProperties properties, byte[] body) throws IOException {
6         try {
7             Thread.sleep(1000); // 处理消息比较慢 一秒处理一个消息
8         } catch (InterruptedException e) {
9             e.printStackTrace();
10        }
11        System.out.println("消费者2: "+new String(body));
12    }
});
```

4. 测试结果

```
消费者1: 1====>:我是消息  
消费者1: 3====>:我是消息  
消费者1: 5====>:我是消息  
消费者1: 7====>:我是消息  
消费者1: 9====>:我是消息
```

```
消费者2: 0====>:我是消息  
消费者2: 2====>:我是消息  
消费者2: 4====>:我是消息  
消费者2: 6====>:我是消息  
消费者2: 8====>:我是消息
```

总结：默认情况下，RabbitMQ将按顺序将每个消息发送给下一个使用者。平均而言，每个消费者都会收到相同数量的消息。这种分发消息的方式称为循环。

5. 消息自动确认机制

Doing a task can take a few seconds. You may wonder what happens if one of the consumers starts a long task and dies with it only partly done. With our current code, once RabbitMQ delivers a message to the consumer it immediately marks it for deletion. In this case, if you kill a worker we will lose the message it was just processing. We'll also lose all the messages that were dispatched to this particular worker but were not yet handled.

But we don't want to lose any tasks. If a worker dies, we'd like the task to be delivered to another worker.

```
1 channel.basicQos(1); //一次只接受一条未确认的消息  
2 //参数2：关闭自动确认消息  
3 channel.basicConsume("hello", false, new DefaultConsumer(channel){  
4     @Override  
5     public void handleDelivery(String consumerTag, Envelope envelope,  
6         AMQP.BasicProperties properties, byte[] body) throws IOException {  
7         System.out.println("消费者1: "+new String(body));  
8         channel.basicAck(envelope.getDeliveryTag(), false); //手动确认消息  
9     }  
});
```

- 设置通道一次只能消费一个消息
- 关闭消息的自动确认,开启手动确认消息

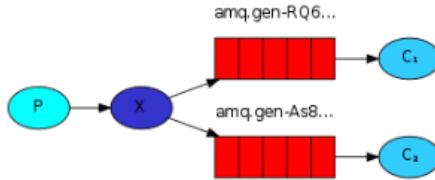
```
消费者1: 0====>:我是消息  
消费者1: 2====>:我是消息  
消费者1: 3====>:我是消息  
消费者1: 4====>:我是消息  
消费者1: 5====>:我是消息  
消费者1: 6====>:我是消息  
消费者1: 7====>:我是消息  
消费者1: 8====>:我是消息  
消费者1: 9====>:我是消息
```

```
消费者2: 1====>:我是消息
```

4.5 第三种模型(fanout)

fanout 扇出 也称为广播

Putting it all together



在广播模式下，消息发送流程是这样的：

- 可以有多个消费者
- 每个消费者有自己的queue（队列）
- 每个队列都要绑定到Exchange（交换机）
- 生产者发送的消息，只能发送到交换机，交换机来决定要发给哪个队列，生产者无法决定。
- 交换机把消息发送给绑定过的所有队列
- 队列的消费者都能拿到消息。实现一条消息被多个消费者消费
- 注：比如用户注册后，不同的消费者获取这个状态信息后，有的发送验证短信，有的发送使用手册，有的发送推广信息等

1. 开发生产者

```
1 //声明交换机
2 //参数1：交换机名称  参数2：交换机的类型，当交换机的类型是fanout的时候，表示是一种广播类型
3 channel.exchangeDeclare("logs", "fanout"); //广播 一条消息多个消费者同时消费
4 //发布消息
5 channel.basicPublish("logs", "", null, "hello".getBytes());
```

2. 开发消费者-1

```
1 //绑定交换机
2 channel.exchangeDeclare("logs", "fanout");
3 //创建临时队列
4 String queue = channel.queueDeclare().getQueue();
5 //将临时队列绑定exchange
6 channel.queueBind(queue, "logs", "");
7 //处理消息
8 channel.basicConsume(queue, true, new DefaultConsumer(channel){
9     @Override
10    public void handleDelivery(String consumerTag, Envelope envelope,
11        AMQP.BasicProperties properties, byte[] body) throws IOException {
12        System.out.println("消费者1：" + new String(body));
13    }
});
```

3. 开发消费者-2

```
1 //绑定交换机
2 channel.exchangeDeclare("logs", "fanout");
3 //创建临时队列
4 String queue = channel.queueDeclare().getQueue();
5 //将临时队列绑定exchange
6 channel.queueBind(queue, "logs", "");
7 //处理消息
8 channel.basicConsume(queue, true, new DefaultConsumer(channel){
9     @Override
10    public void handleDelivery(String consumerTag, Envelope envelope,
11        AMQP.BasicProperties properties, byte[] body) throws IOException {
12        System.out.println("消费者2: "+new String(body));
13    }
14});
```

4. 开发消费者-3

```
1 //绑定交换机
2 channel.exchangeDeclare("logs", "fanout");
3 //创建临时队列
4 String queue = channel.queueDeclare().getQueue();
5 //将临时队列绑定exchange
6 channel.queueBind(queue, "logs", "");
7 //处理消息
8 channel.basicConsume(queue, true, new DefaultConsumer(channel){
9     @Override
10    public void handleDelivery(String consumerTag, Envelope envelope,
11        AMQP.BasicProperties properties, byte[] body) throws IOException {
12        System.out.println("消费者3: "+new String(body));
13    }
14});
```

5. 测试结果

The image shows three separate terminal windows from a Java IDE, each running a different consumer class. Each window has a title bar labeled 'Run:' followed by the consumer name and '(1)'.

- The first window shows the output: 消费者1: hello
- The second window shows the output: 消费者2: hello
- The third window shows the output: 消费者3: hello

The outputs are identical, indicating that all three consumers are receiving the same message from the provider.

4.6 第四种模型(Routing)

4.6.1 Routing 之订阅模型-Direct(直连)

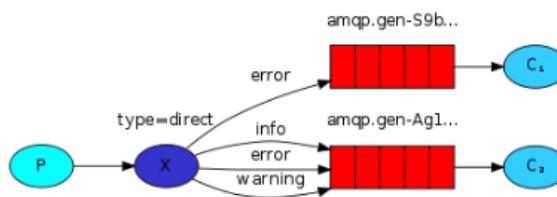
在Fanout模式中，一条消息，会被所有订阅的队列都消费。但是，在某些场景下，我们希望不同的消息被不同的队列消费。这时就要用到Direct类型的Exchange。

在Direct模型下：

- 队列与交换机的绑定，不能是任意绑定了，而是要指定一个 `RoutingKey` (路由key)
- 消息的发送方在向 Exchange发送消息时，也必须指定消息的 `RoutingKey`。
- Exchange不再把消息交给每一个绑定的队列，而是根据消息的 `Routing Key` 进行判断，只有队列的 `Routingkey` 与消息的 `Routing key` 完全一致，才会接收到消息

流程：

Putting it all together



图解：

- P: 生产者，向Exchange发送消息，发送消息时，会指定一个routing key。
- X: Exchange (交换机)，接收生产者的消息，然后把消息递交给与routing key完全匹配的队列
- C1: 消费者，其所在队列指定了需要routing key为 error 的消息
- C2: 消费者，其所在队列指定了需要routing key为 info、error、warning 的消息

1. 开发生产者

```
1 //声明交换机 参数1:交换机名称 参数2:交换机类型 基于指令的Routing key转发
2 channel.exchangeDeclare("logs_direct", "direct");
3 String key = "";
4 //发布消息
5 channel.basicPublish("logs_direct", key, null, ("指定的route key"+key+"的消息").getBytes());
```

2.开发消费者-1

```
1 //声明交换机
2 channel.exchangeDeclare("logs_direct", "direct");
3 //创建临时队列
4 String queue = channel.queueDeclare().getQueue();
5 //绑定队列和交换机
6 channel.queueBind(queue, "logs_direct", "error");
7 channel.queueBind(queue, "logs_direct", "info");
8 channel.queueBind(queue, "logs_direct", "warn");
```

```
9 //消费消息
10 channel.basicConsume(queue,true,new DefaultConsumer(channel){
11     @Override
12     public void handleDelivery(String consumerTag, Envelope envelope,
13         AMQP.BasicProperties properties, byte[] body) throws IOException {
14         System.out.println("消费者1: "+new String(body));
15     }
16 });

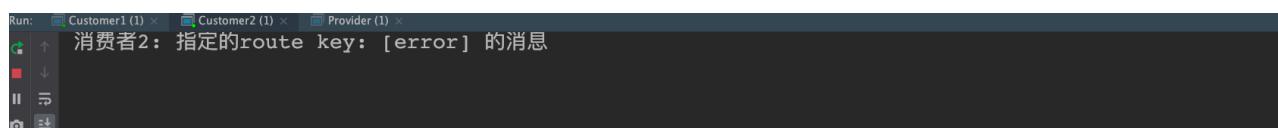
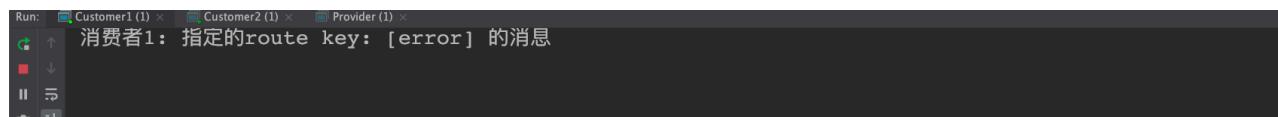
```

3.开发消费者-2

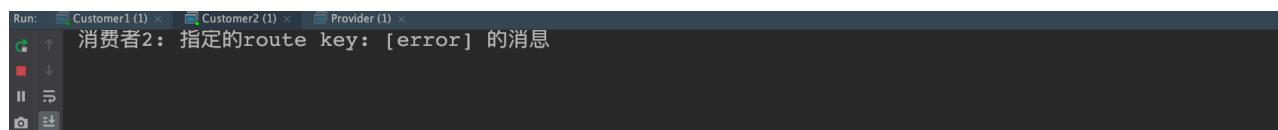
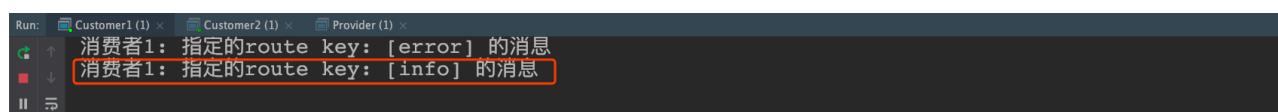
```
1 //声明交换机
2 channel.exchangeDeclare("logs_direct","direct");
3 //创建临时队列
4 String queue = channel.queueDeclare().getQueue();
5 //绑定队列和交换机
6 channel.queueBind(queue,"logs_direct","error");
7 //消费消息
8 channel.basicConsume(queue,true,new DefaultConsumer(channel){
9     @Override
10    public void handleDelivery(String consumerTag, Envelope envelope,
11        AMQP.BasicProperties properties, byte[] body) throws IOException {
12        System.out.println("消费者2: "+new String(body));
13    }
14 });

```

4.测试生产者发送Route key为error的消息时

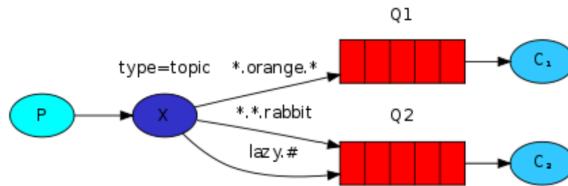


5.测试生产者发送Route key为info的消息时



4.6.2 Routing 之订阅模型-Topic

Topic 类型的 Exchange 与 Direct 相比，都是可以根据 RoutingKey 把消息路由到不同的队列。只不过 Topic 类型 Exchange 可以让队列在绑定 Routing key 的时候使用通配符！这种模型 Routingkey 一般都是由一个或多个单词组成，多个单词之间以“.”分割，例如： item.insert



```
1 # 统配符
2     * (star) can substitute for exactly one word. 匹配不多不少恰好1个词
3     # (hash) can substitute for zero or more words. 匹配一个或多个词
4 # 如：
5     audit.# 匹配audit.irs.corporate或者 audit.irs 等
6     audit.* 只能匹配 audit.irs
```

1. 开发生产者

```
1 //生命交换机和交换机类型 topic 使用动态路由(通配符方式)
2 channel.exchangeDeclare("topics","topic");
3 String routekey = "user.save"; //动态路由key
4 //发布消息
5 channel.basicPublish("topics",routekey,null,("这是路由中的动态订阅模型,route
key: ["+routekey+"]").getBytes());
```

2. 开发消费者-1

Routing Key 中使用*通配符方式

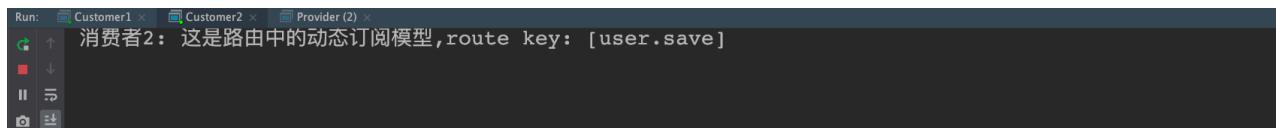
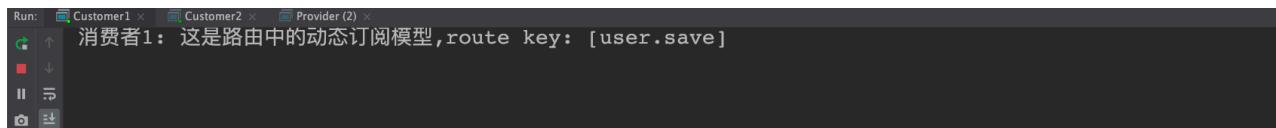
```
1 //声明交换机
2 channel.exchangeDeclare("topics","topic");
3 //创建临时队列
4 String queue = channel.queueDeclare().getQueue();
5 //绑定队列与交换机并设置获取交换机中动态路由
6 channel.queueBind(queue,"topics","user.*");
7
8 //消费消息
9 channel.basicConsume(queue,true,new DefaultConsumer(channel){
10     @Override
11     public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
12         System.out.println("消费者1: "+new String(body));
13     }
14});
```

3. 开发消费者-2

Routing Key中使用#通配符方式

```
1 //声明交换机
2 channel.exchangeDeclare("topics", "topic");
3 //创建临时队列
4 String queue = channel.queueDeclare().getQueue();
5 //绑定队列与交换机并设置获取交换机中动态路由
6 channel.queueBind(queue, "topics", "user.#");
7
8 //消费消息
9 channel.basicConsume(queue, true, new DefaultConsumer(channel){
10     @Override
11     public void handleDelivery(String consumerTag, Envelope envelope,
12         AMQP.BasicProperties properties, byte[] body) throws IOException {
13         System.out.println("消费者2: "+new String(body));
14     }
15});
```

4. 测试结果



5. SpringBoot中使用RabbitMQ

5.0 搭建初始环境

1. 引入依赖

```
1 <dependency>
2     <groupId>org.springframework.boot</groupId>
3     <artifactId>spring-boot-starter-amqp</artifactId>
4 </dependency>
```

2. 配置配置文件

```
1 spring:
2   application:
3     name: springboot_rabbitmq
4   rabbitmq:
5     host: 10.15.0.9
6     port: 5672
7     username: ems
8     password: 123
9     virtual-host: /ems
```

RabbitTemplate 用来简化操作 使用时候直接在项目中注入即可使用

5.1 第一种hello world模型使用

1. 开发生产者

```
1 @Autowired
2 private RabbitTemplate rabbitTemplate;
3
4 @Test
5 public void testHello(){
6   rabbitTemplate.convertAndSend("hello", "hello world");
7 }
```

2. 开发消费者

```
1 @Component
2 @RabbitListener(queuesToDeclare = @Queue("hello"))
3 public class HelloCustomer {
4
5   @RabbitHandler
6   public void receive1(String message){
7     System.out.println("message = " + message);
8   }
9 }
```

5.2 第二种work模型使用

1. 开发生产者

```

1  @Autowired
2  private RabbitTemplate rabbitTemplate;
3
4  @Test
5  public void testWork(){
6      for (int i = 0; i < 10; i++) {
7          rabbitTemplate.convertAndSend("work", "hello work!");
8      }
9  }

```

2. 开发消费者

```

1  @Component
2  public class WorkCustomer {
3      @RabbitListener(queuesToDeclare = @Queue("work"))
4      public void receive1(String message){
5          System.out.println("work message1 = " + message);
6      }
7
8      @RabbitListener(queuesToDeclare = @Queue("work"))
9      public void receive2(String message){
10         System.out.println("work message2 = " + message);
11     }
12 }

```

说明：默认在Spring AMQP实现中work这种方式就是公平调度，如果需要实现能者多劳需要额外配置

5.3 Fanout 广播模型

1. 开发生产者

```

1  @Autowired
2  private RabbitTemplate rabbitTemplate;
3
4  @Test
5  public void testFanout() throws InterruptedException {
6      rabbitTemplate.convertAndSend("logs", "", "这是日志广播");
7  }

```

2. 开发消费者

```

1  @Component
2  public class FanoutCustomer {
3
4      @RabbitListener(bindings = @QueueBinding(

```

```

5         value = @Queue,
6         exchange = @Exchange(name="logs",type = "fanout")
7     ))
8     public void receive1(String message){
9         System.out.println("message1 = " + message);
10    }
11
12     @RabbitListener(bindings = @QueueBinding(
13             value = @Queue, //创建临时队列
14             exchange = @Exchange(name="logs",type = "fanout") //绑定交
换机类型
15         ))
16     public void receive2(String message){
17         System.out.println("message2 = " + message);
18     }
19 }
```

5.4 Route 路由模型

1. 开发生产者

```

1 @Autowired
2 private RabbitTemplate rabbitTemplate;
3
4 @Test
5 public void testDirect(){
6     rabbitTemplate.convertAndSend("directs","error","error 的日志信息");
7 }
```

2. 开发消费者

```

1 @Component
2 public class DirectCustomer {
3
4     @RabbitListener(bindings ={
5             @QueueBinding(
6                 value = @Queue(),
7                 key={"info","error"},
8                 exchange = @Exchange(type =
"direct",name="directs")
9             )})
10    public void receive1(String message){
11        System.out.println("message1 = " + message);
12    }
13
14    @RabbitListener(bindings ={
```

```

15         @QueueBinding(
16             value = @Queue(),
17             key={"error"} ,
18             exchange = @Exchange(type =
19 "direct",name="directs")
20         )))
21     public void receive2(String message){
22         System.out.println("message2 = " + message);
23     }
24 }
```

5.5 Topic 订阅模型(动态路由模型)

1. 开发生产者

```

1 @Autowired
2 private RabbitTemplate rabbitTemplate;
3
4 //topic
5 @Test
6 public void testTopic(){
7     rabbitTemplate.convertAndSend("topics","user.save.findAll","user.save.
8     findAll 的消息");
9 }
```

2. 开发消费者

```

1 @Component
2 public class TopCustomer {
3     @RabbitListener(bindings = {
4         @QueueBinding(
5             value = @Queue,
6             key = {"user.*"} ,
7             exchange = @Exchange(type = "topic",name =
8 "topics")
9         )
10    })
11    public void receive1(String message){
12        System.out.println("message1 = " + message);
13    }
14    @RabbitListener(bindings = {
15        @QueueBinding(
16            value = @Queue,
17            key = {"user.#"},
```

```

18         exchange = @Exchange(type = "topic", name =
19             "topics")
20     })
21     public void receive2(String message){
22         System.out.println("message2 = " + message);
23     }
24 }

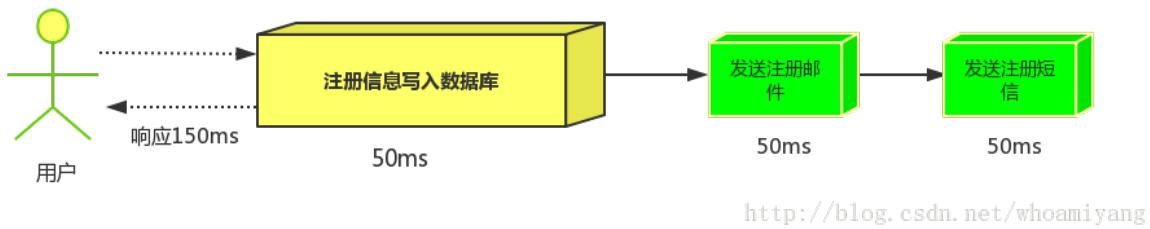
```

6. MQ的应用场景

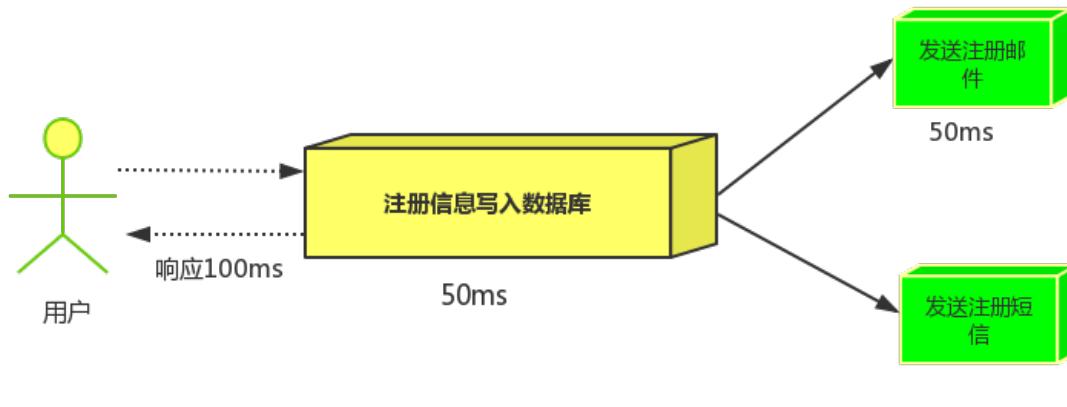
6.1 异步处理

场景说明：用户注册后，需要发注册邮件和注册短信，传统的做法有两种 1.串行的方式 2.并行的方式

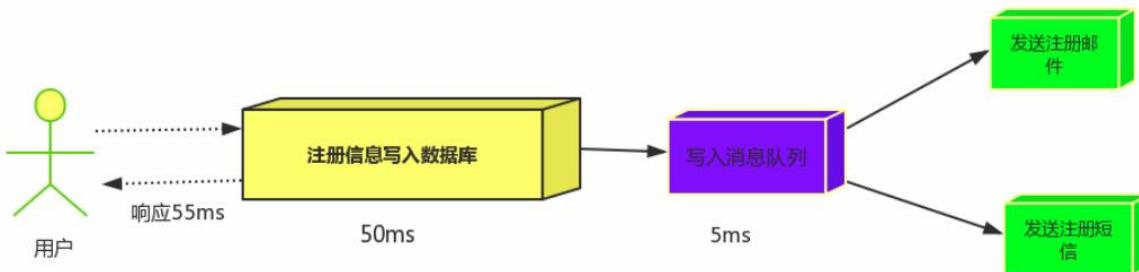
- 串行方式：**将注册信息写入数据库后,发送注册邮件,再发送注册短信,以上三个任务全部完成后才返回给客户端。这有一个问题是,邮件,短信并不是必须的,它只是一个通知,而这种做法让客户端等待没有必要等待的东西.



- 并行方式：**将注册信息写入数据库后,发送邮件的同时,发送短信,以上三个任务完成后,返回给客户端,并行的方式能提高处理的时间。



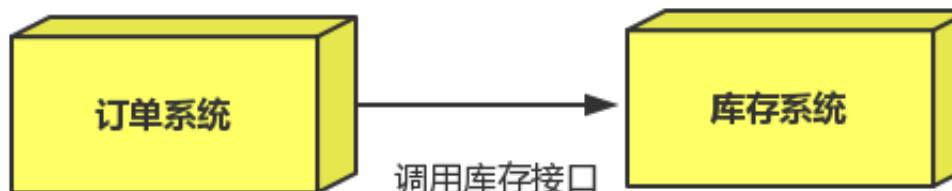
- 消息队列：假设三个业务节点分别使用50ms,串行方式使用时间150ms,并行使用时间100ms。虽然并行已经提高的处理时间,但是,前面说过,邮件和短信对我正常的使用网站没有任何影响,客户端没有必要等着其发送完成才显示注册成功,应该是写入数据库后就返回。消息队列：引入消息队列后,把发送邮件,短信不是必须的业务逻辑异步处理



由此可以看出,引入消息队列后,用户的响应时间就等于写入数据库的时间+写入消息队列的时间(可以忽略不计),引入消息队列后处理后,响应时间是串行的3倍,是并行的2倍。

6.2 应用解耦

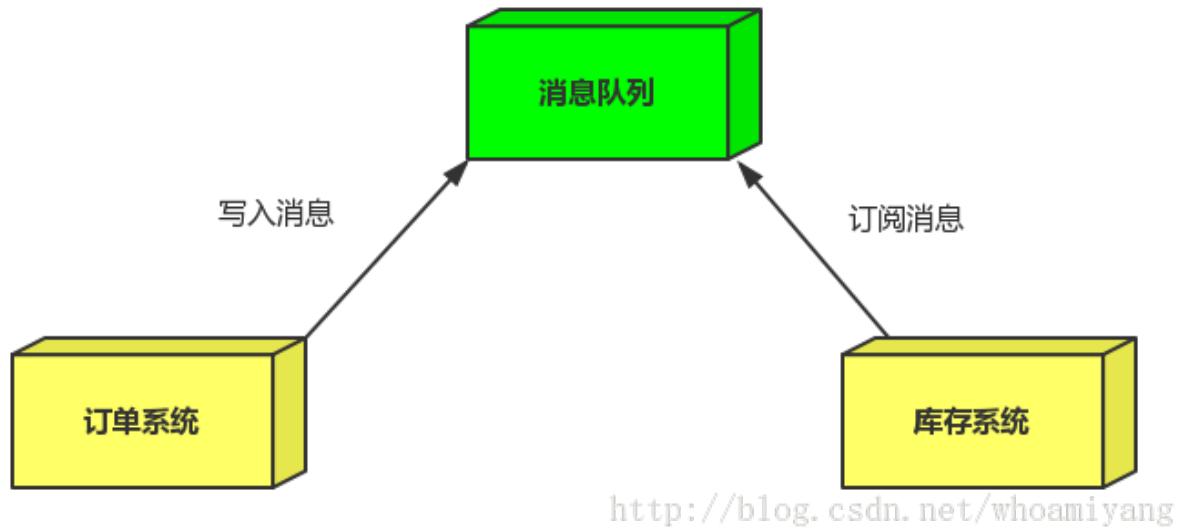
场景：双11是购物狂节,用户下单后,订单系统需要通知库存系统,传统的做法就是订单系统调用库存系统的接口。



<http://blog.csdn.net/whoamiyang>

这种做法有一个缺点:

当库存系统出现故障时,订单就会失败。订单系统和库存系统高耦合. 引入消息队列



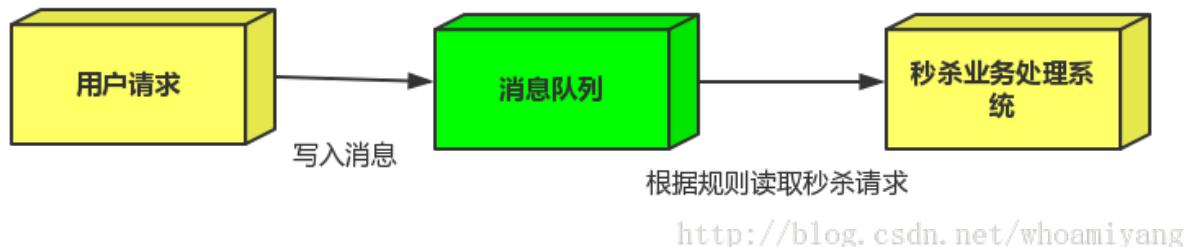
- **订单系统:** 用户下单后,订单系统完成持久化处理,将消息写入消息队列,返回用户订单下单成功。
- **库存系统:** 订阅下单的消息,获取下单消息,进行库操作。就算库存系统出现故障,消息队列也能保证消息的可靠投递,不会导致消息丢失.

6.3 流量削峰

场景: 秒杀活动, 一般会因为流量过大, 导致应用挂掉, 为了解决这个问题, 一般在应用前端加入消息队列。

作用:

- 1.可以控制活动人数, 超过此一定阀值的订单直接丢弃(我为什么秒杀一次都没有成功过呢^^)
- 2.可以缓解短时间的高流量压垮应用(应用程序按自己的最大处理能力获取订单)



- 1.用户的请求,服务器收到之后,首先写入消息队列,加入消息队列长度超过最大值,则直接抛弃用户请求或跳转到错误页面.
- 2.秒杀业务根据消息队列中的请求信息, 再做后续处理.

7. RabbitMQ的集群

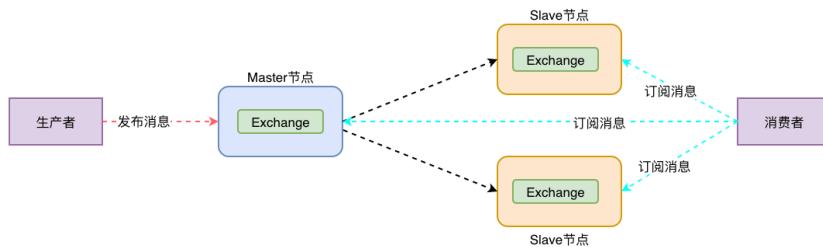
7.1 集群架构

7.1.1 普通集群(副本集群)

All data/state required for the operation of a RabbitMQ broker is replicated across all nodes. An exception to this are message queues, which by default reside on one node, though they are visible and reachable from all nodes. To replicate queues across nodes in a cluster --摘自官网

默认情况下: RabbitMQ代理操作所需的所有数据/状态都将跨所有节点复制。这方面的一个例外是消息队列, 默认情况下, 消息队列位于一个节点上, 尽管它们可以从所有节点看到和访问

1. 架构图



核心解决问题: 当集群中某一时刻master节点宕机, 可以对Quene中信息, 进行备份

2. 集群搭建

```
1 # 0.集群规划
2     node1: 10.15.0.3 mq1 master 主节点
3     node2: 10.15.0.4 mq2 repl1 副本节点
4     node3: 10.15.0.5 mq3 repl2 副本节点
5
6 # 1.克隆三台机器主机名和ip映射
7     vim /etc/hosts加入:
8         10.15.0.3 mq1
9         10.15.0.4 mq2
10        10.15.0.5 mq3
11    node1: vim /etc/hostname 加入: mq1
12    node2: vim /etc/hostname 加入: mq2
13    node3: vim /etc/hostname 加入: mq3
14
15 # 2.三个机器安装rabbitmq,并同步cookie文件,在node1上执行:
16     scp /var/lib/rabbitmq/.erlang.cookie root@mq2:/var/lib/rabbitmq/
17     scp /var/lib/rabbitmq/.erlang.cookie root@mq3:/var/lib/rabbitmq/
18
19 # 3.查看cookie是否一致:
20     node1: cat /var/lib/rabbitmq/.erlang.cookie
21     node2: cat /var/lib/rabbitmq/.erlang.cookie
22     node3: cat /var/lib/rabbitmq/.erlang.cookie
23
24 # 4.后台启动rabbitmq所有节点执行如下命令,启动成功访问管理界面:
25     rabbitmq-server -detached
```

```

26
27 # 5.在node2和node3执行加入集群命令:
28     1.关闭      rabbitmqctl stop_app
29     2.加入集群    rabbitmqctl join_cluster rabbit@mq1
30     3.启动服务    rabbitmqctl start_app
31
32 # 6.查看集群状态,任意节点执行:
33     rabbitmqctl cluster_status
34
35 # 7.如果出现如下显示,集群搭建成功:
36     Cluster status of node rabbit@mq3 ...
37     [{nodes,[{disc,[rabbit@mq1,rabbit@mq2,rabbit@mq3]}]},,
38     {running_nodes,[rabbit@mq1,rabbit@mq2,rabbit@mq3]},,
39     {cluster_name,<<"rabbit@mq1">>},,
40     {partitions,[]},,
41     {alarms,[{rabbit@mq1,[]},{rabbit@mq2,[]},{rabbit@mq3,[]}]}]
42
43 # 8.登录管理界面,展示如下状态:

```

The screenshot shows the RabbitMQ Management Interface's Overview page. It displays the following cluster status:

Name	File descriptors	Socket descriptors	Erlang processes	Memory	Disk space	Uptime	Info	Reset stats
rabbit@mq1	38 1024 available	0 829 available	396 1048576 available	79MB 390MB high watermark	16GiB 48MB low watermark	32m 14s	basic disc 1 rss	This node All nodes
rabbit@mq2	36 1024 available	1 829 available	405 1048576 available	77MB 390MB high watermark	16GiB 48MB low watermark	31m 35s	basic disc 1 rss	This node All nodes
rabbit@mq3	37 1024 available	0 829 available	394 1048576 available	77MB 390MB high watermark	16GiB 48MB low watermark	30m 56s	basic disc 1 rss	This node All nodes

1 # 9. 测试集群在node1上, 创建队列

The screenshot shows the RabbitMQ Management Interface's Queues page. It displays the following queue information:

Name	Node	Features	State	Messages	Message rates
hello	rabbit@mq1	D	idle	Ready: 0, Unacked: 0, Total: 0	incoming: 0, deliver / get: 0, ack: 0

1 # 10. 查看node2和node3节点:

The screenshot shows the RabbitMQ Management Interface's Queues page for the mq2 and mq3 nodes. It displays the following queue information:

Name	Node	Features	State	Messages	Message rates
hello	rabbit@mq1	D	idle	Ready: 0, Unacked: 0, Total: 0	incoming: 0, deliver / get: 0, ack: 0

Queues

All queues (1)

Name	Node	Features	State	Ready	Unacked	Total	Message rates
hello	rabbit@mq1	D	idle	0	0	0	incoming deliver / get ack

Queues

All queues (1)

Name	Node	Features	State	Ready	Unacked	Total	Message rates
hello	rabbit@mq1	D	down	NaN	NaN	NaN	incoming deliver / get ack

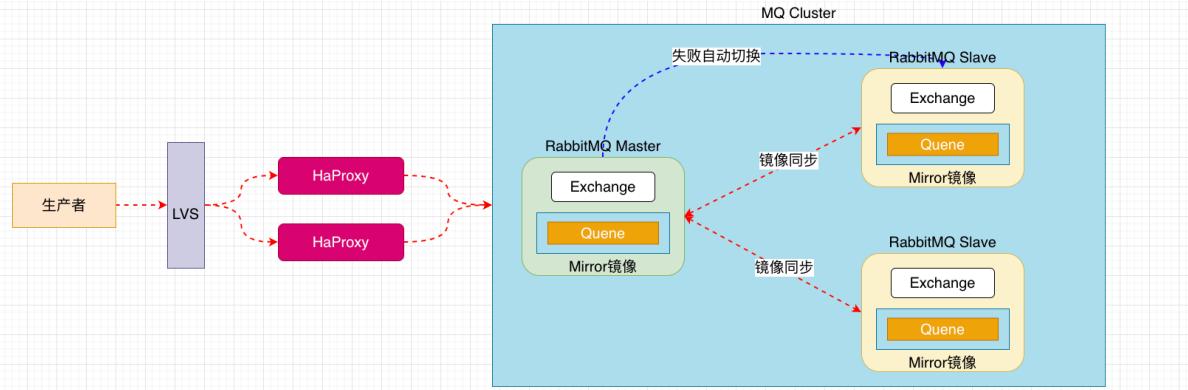
7.1.2 镜像集群

This guide covers mirroring (queue contents replication) of classic queues --摘自官网

By default, contents of a queue within a RabbitMQ cluster are located on a single node (the node on which the queue was declared). This is in contrast to exchanges and bindings, which can always be considered to be on all nodes. Queues can optionally be made *mirrored* across multiple nodes. --摘自官网

镜像队列机制就是将队列在三个节点之间设置主从关系，消息会在三个节点之间进行自动同步，且如果其中一个节点不可用，并不会导致消息丢失或服务不可用的情况，提升MQ集群的整体高可用性。

1. 集群架构图



2. 配置集群架构

```

1 # 0.策略说明
2     rabbitmqctl set_policy [-p <vhost>] [--priority <priority>] [--apply-to <apply-to>] <name> <pattern> <definition>
3         -p Vhost: 可选参数, 针对指定vhost下的queue进行设置
4             Name: policy的名称
5             Pattern: queue的匹配模式(正则表达式)
6             Definition: 镜像定义, 包括三个部分ha-mode, ha-params, ha-sync-mode
7                 ha-mode: 指明镜像队列的模式, 有效值为 all/exactly/nodes
8                     all: 表示在集群中所有的节点上进行镜像
9                     exactly: 表示在指定个数的节点上进行镜像, 节点的个数由
10                ha-params指定
11
12                nodes: 表示在指定的节点上进行镜像, 节点名称通过ha-
13            params指定
14
15            ha-params: ha-mode模式需要用到的参数
16            ha-sync-mode: 进行队列中消息的同步方式, 有效值为automatic和
17            manual
18
19 # 1.查看当前策略
20     rabbitmqctl list_policies
21
22 # 2.添加策略
23     rabbitmqctl set_policy ha-all '^hello' '{"ha-mode":"all","ha-sync-
24     mode":"automatic"}'
25         说明:策略正则表达式为 “^” 表示所有匹配所有队列名称 ^hello:匹配hello开头队列
26
27 # 3.删除策略
28     rabbitmqctl clear_policy ha-all
29
30 # 4.测试集群

```

问题记录：

使用docker安装rabbitmq

```
1 | docker run --name rabbitmq -d -p 15672:15672 -p 5672:5672 -e  
| RABBITMQ_DEFAULT_USER=liyang -e RABBITMQ_DEFAULT_PASS=liyang  
| rabbitmq:management
```

消费者消费信息是在子线程完成的?

如果主线程结束或者使用junit，就会导致主线程接受不到回调，因此消费了消息，但是主线程收不到回调的信息！

@Test不支持多线程吗？