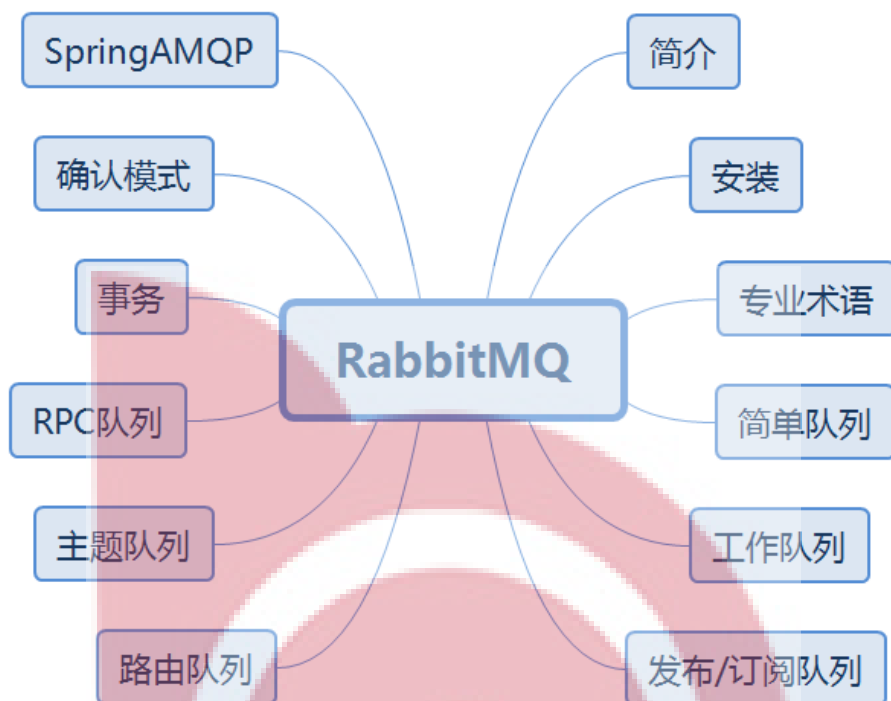




RabbitMQ

学习目标



MQ简介

在计算机科学中，消息队列（英语：Message queue）是一种进程间通信或同一进程的不同线程间的通信方式，软件的队列用来处理一系列的输入，通常是来自用户。消息队列提供了异步的通信协议，每一个队列中的纪录包含详细说明的数据，包含发生的时间，输入设备的种类，以及特定的输入参数，也就是说：消息的发送者和接收者不需要同时与消息队列交互。消息会保存在队列中，直到接收者取回它。

实现

消息队列常常保存在链表结构中。拥有权限的进程可以向消息队列中写入或读取消息。

目前，有很多消息队列有很多开源的实现，包括 JBoss Messaging、JORM、Apache ActiveMQ、Sun Open Message Queue、IBM MQ、Apache Qpid 和 HTTPSQS。

当前使用较多的消息队列有 RabbitMQ、RocketMQ、ActiveMQ、Kafka、ZeroMQ、MetaMq 等，而部分数据库如 Redis、MySQL 以及 phxsql 也可实现消息队列的功能。

特点

MQ是消费者-生产者模型的一个典型的代表，一端往消息队列中不断写入消息，而另一端则可以读取或者订阅队列中的消息。MQ和JMS类似，但不同的是JMS是SUN JAVA消息中间件服务的一个标准和API定义，而MQ则是遵循了AMQP协议的具体实现和产品。

注意：



1. **AMQP**，即Advanced Message Queuing Protocol,一个提供统一消息服务的应用层标准高级消息队列协议,是应用层协议的一个开放标准,为面向消息的中间件设计。基于此协议的客户端与消息中间件可传递消息，并不受客户端/中间件不同产品，不同的开发语言等条件的限制。

2. **JMS**，Java消息服务（Java Message Service）应用程序接口，是一个Java平台中关于面向消息中间件的API，用于在两个应用程序之间，或分布式系统中发送消息，进行异步通信。Java消息服务是一个与具体平台无关的API，绝大多数MOM提供商都对JMS提供支持。常见的消息队列，大部分都实现了JMS API，如 **ActiveMQ**，**Redis** 以及 **RabbitMQ** 等。

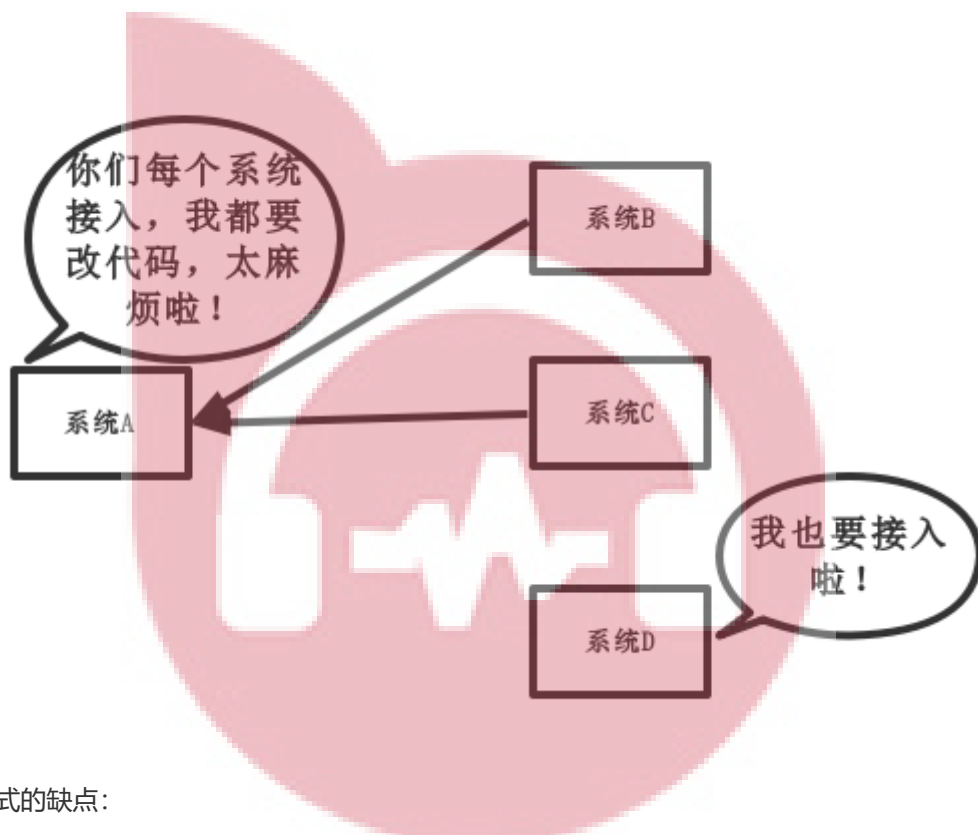
优缺点

优点

应用耦合、异步处理、流量削锋

- 解耦

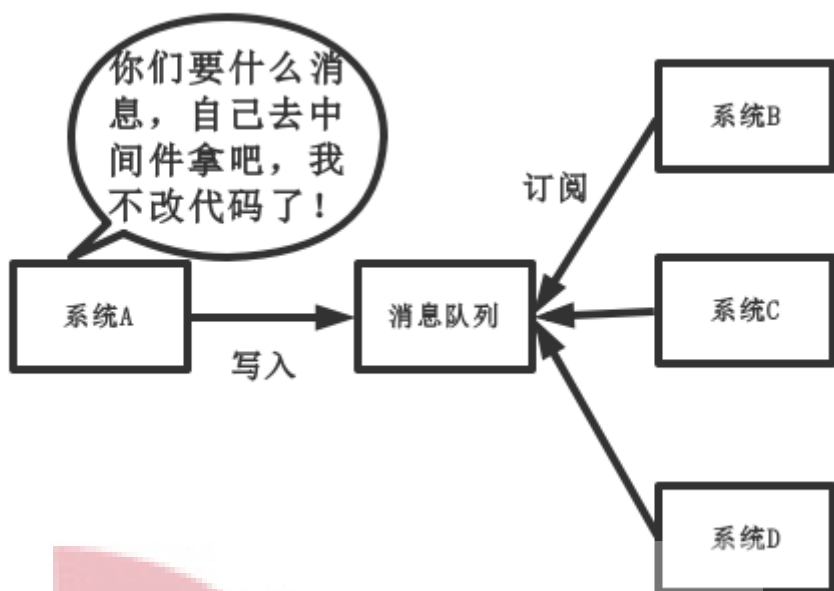
传统模式：



传统模式的缺点：

系统间耦合性太强，如上图所示，系统A在代码中直接调用系统B和系统C的代码，如果将来D系统接入，系统A还需要修改代码，过于麻烦！

中间件模式

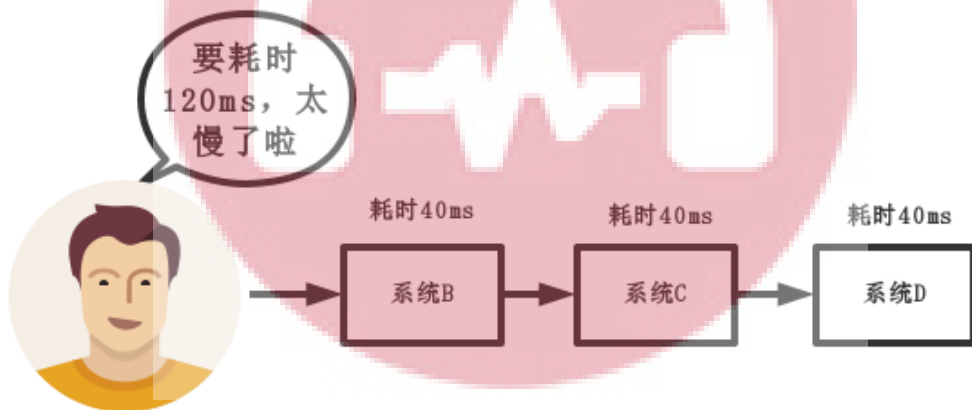


中间件模式的的优点：

将消息写入消息队列，需要消息的系统自己从消息队列中订阅，从而系统A不需要做任何修改。

- 异步

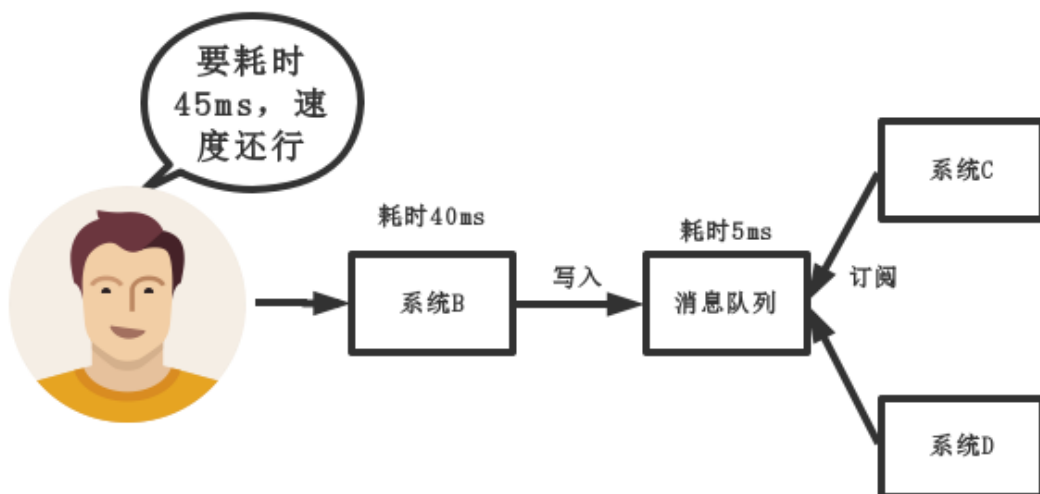
传统模式：



传统模式的缺点：

一些非必要的业务逻辑以同步的方式运行，太耗费时间。

中间件模式：



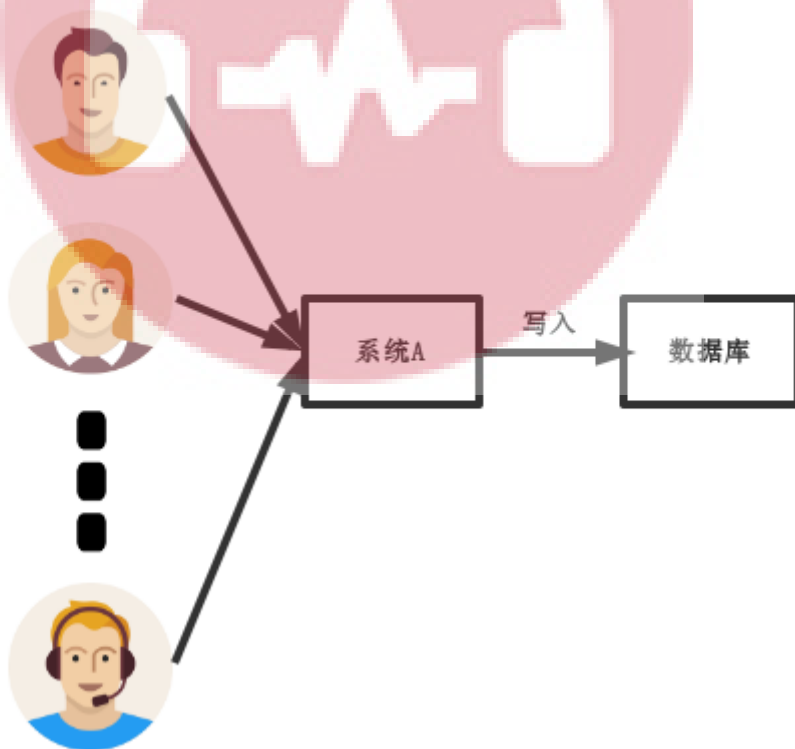
中间件模式的的优点:

将消息写入消息队列，需要消息的系统自己从消息队列中订阅，从而系统A不需要做任何修改。

- 削峰

传统模式:

2000个用户同时请求



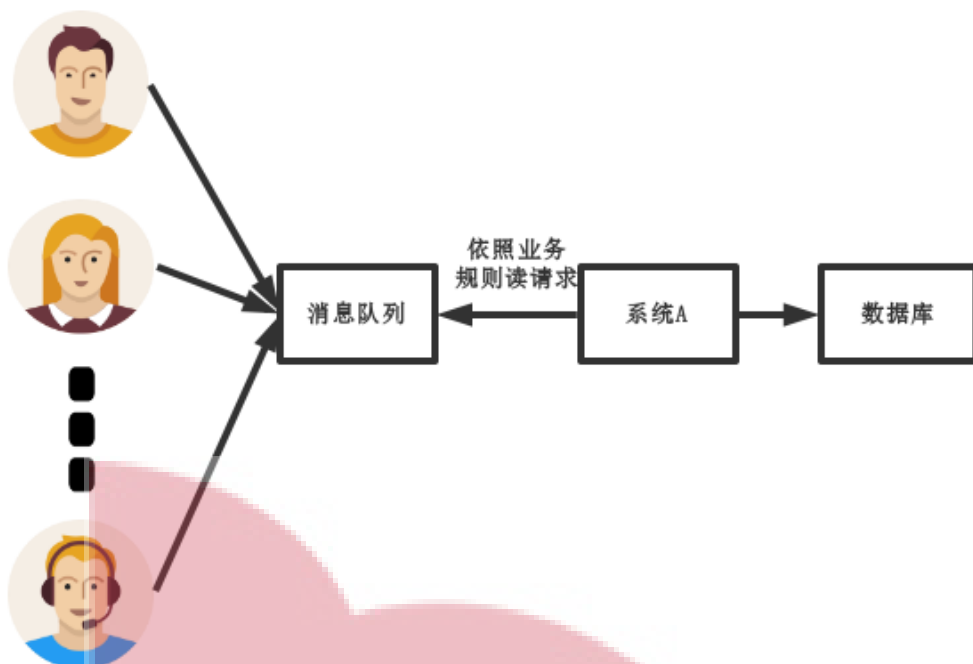
传统模式的缺点:

并发量大的时候，所有的请求直接怼到数据库，造成数据库连接异常

中间件模式:



2000个用户同时请求



中间件模式的的优点：

系统A慢慢的按照数据库能处理的并发量，从消息队列中慢慢拉取消息。在生产中，这个短暂的高峰期积压是允许的。

缺点

系统可用性降低、系统复杂性增加

使用场景

消息队列，是分布式系统中重要的组件，其通用的使用场景可以简单地描述为：**当不需要立即获得结果，但是并发量又需要进行控制的时候，差不多就是需要使用消息队列的时候**

在项目中，将一些无需即时返回且耗时的操作提取出来，进行了异步处理，而这种异步处理的方式大大的节省了服务器的请求响应时间，从而提高了系统的吞吐量。

为什么使用RabbitMQ

AMQP，即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。消息中间件主要用于组件之间的解耦，消息的发送者无需知道消息使用者的存在，反之亦然。

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。

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，如：Python、Ruby、.NET、Java、JMS、C、PHP、ActionScript、XMPP、STOMP等，支持AJAX。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。

总结如下：

- 基于AMQP协议
- 高并发(是一个容量的概念，服务器可以接受的最大任务数量)
- 高性能(是一个速度的概念，单位时间内服务器可以处理的任務数)



—— 程序员乐字节 (是一个持久的概念, 单位时间内服务器可以正常工作的时间比例)

- 强大的社区支持, 以及很多公司都在使用
- 支持插件
- 支持多语言

安装

安装Erlang

官网提示: <https://www.erlang-solutions.com/resources/download.html>

23.0.2	
CentOS 8 (arm64-bit)	DOWNLOAD
CentOS 8 (64-bit)	DOWNLOAD
CentOS 7 (arm64-bit)	DOWNLOAD
CentOS 7 (64-bit)	DOWNLOAD
CentOS 6 (64-bit)	DOWNLOAD

安装erlang

```
yum -y install esl-erlang_23.0.2-1_centos_7_amd64.rpm
```

检测erlang

```
[root@localhost ~]# erl
Erlang/OTP 23 [erts-11.0.2] [source] [64-bit] [smp:1:1] [ds:1:1:10] [async-threads:1] [hipe]
Eshell V11.0.2 (abort with ^G)
1>
```

安装RabbitMQ

文件下载

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

Downloading and Installing RabbitMQ

The latest [release](#) of RabbitMQ is **3.8.5**. See [change log](#) for release notes. See [RabbitMQ support timeline](#) to find out what release series are supported.

Experimenting with RabbitMQ on your workstation? Try the [community Docker image](#):

```
docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:3-management
```

RabbitMQ Server

Installation Guides

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

Kubernetes

- VMware [Tanzu RabbitMQ on Kubernetes](#) (currently in beta)
- A [peer discovery](#) mechanism [for Kubernetes](#)
- [Examples](#) that use Minikube and Kind

In This Section

- [Install: Windows](#)
- [Install: Debian and Ubuntu](#)
- [Install: RPM-based Linux](#)
- [Install: Homebrew](#)
- [Install: Windows \(manual\)](#)
- [Install: Generic binary build](#)
- [Install: Solaris](#)
- [Install: EC2](#)
- [Upgrade](#)
- [Blue-green deployment: based upgrade](#)
- [Supported Platforms](#)
- [Changelog](#)
- [Erlang Versions](#)
- [Signed Packages](#)
- [Java Client Downloads](#)
- [.NET Client Downloads](#)
- [Erlang Client Downloads](#)
- [Community Plugins](#)
- [Snapshots](#)

安装rabbitmq



— 程序员乐字节 — `install rabbitmq-server-3.8.5-1.el7.noarch.rpm`

安装UI插件

```
rabbitmq-plugins enable rabbitmq_management
```

```
[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.
```

启用rabbitmq服务

```
systemctl start rabbitmq-server.service
```

检测服务

```
systemctl status rabbitmq-server.service
```

```
[root@localhost ~]# systemctl status rabbitmq-server.service
● rabbitmq-server.service - RabbitMQ broker
   Loaded: loaded (/usr/lib/systemd/system/rabbitmq-server.service; disabled; vendor preset: disabled)
   Active: active (running) since 2020-07-04 16:49:59 CST; 39s ago
   Main PID: 2026 (beam.smp)
   Status: "Initialized"
   CGroup: /system.slice/rabbitmq-server.service
           └─2026 /usr/lib/erlang/erts-11.0.2/bin/beam.smp -W w -K true -A 64 -MBas ageffcbf -MHAs ageffcbf -MBlmcs 512 -MHlmbcs 512 -MMmcs 30 -P ...
             └─2127 erl_child_setup 32768
               └─2167 inet_gethost 4
                 └─2168 inet_gethost 4

7# 04 16:49:57 localhost.localdomain rabbitmq-server[2026]: ##### Licensed under the MPL 1.1. Website: https://rabbitmq.com
7# 04 16:49:57 localhost.localdomain rabbitmq-server[2026]: Doc guides: https://rabbitmq.com/documentation.html
7# 04 16:49:57 localhost.localdomain rabbitmq-server[2026]: Support: https://rabbitmq.com/contact.html
7# 04 16:49:57 localhost.localdomain rabbitmq-server[2026]: Tutorials: https://rabbitmq.com/getstarted.html
7# 04 16:49:57 localhost.localdomain rabbitmq-server[2026]: Monitoring: https://rabbitmq.com/monitoring.html
7# 04 16:49:57 localhost.localdomain rabbitmq-server[2026]: Logs: /var/log/rabbitmq/rabbit@localhost.log
7# 04 16:49:57 localhost.localdomain rabbitmq-server[2026]: /var/log/rabbitmq/rabbit@localhost_upgrade.log
7# 04 16:49:57 localhost.localdomain rabbitmq-server[2026]: Config file(s): (none)
7# 04 16:49:59 localhost.localdomain systemd[1]: Started RabbitMQ broker.
7# 04 16:49:59 localhost.localdomain rabbitmq-server[2026]: Starting broker... completed with 3 plugins.
```

修改防火墙，添加规则

```
-A INPUT -p tcp -m state --state NEW -m tcp --dport 15672 -j ACCEPT
-A INPUT -p tcp -m state --state NEW -m tcp --dport 5672 -j ACCEPT
```

重启防火墙

```
systemctl restart iptables.service
```

访问

guest用户默认只可以localhost(本机)访问



RabbitMQ

Login failed

Username:

Password:

Login

User can only log in via localhost

Close

在rabbitmq的配置文件目录下（默认为：/etc/rabbitmq）创建一个**rabbitmq.config**文件。

文件中添加如下配置（请不要忘记那个“.”）：

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

重启rabbitmq服务

```
systemctl restart rabbitmq-server.service
```

重新访问

RabbitMQ

3.7.12 Erlang 21.1.2

Refreshed 2019-03-17 19:35:27 Refresh every 5 seconds

Virtual host All Cluster rabbit@zk1 User guest Log out

Overview Connections Channels Exchanges Queues Admin

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@zk1	72 32768 available	0 29399 available	391 1048576 available	70MB 391MB high watermark 48MB low watermark	11GB	0m 18s	basic disc 1 rss	This node All nodes	

Churn statistics

Ports and contexts

Export definitions

管理界面基本操作

管理界面添加用户与权限分配

RabbitMQ默认提供guest用户，密码为guest用于登录MQ主页面

RabbitMQ Management

127.0.0.1:15672/#/

RabbitMQ

3.7.12 Erlang 21.2

Refreshed 2019-03-03 15:11:48 Refresh every 5 seconds

Virtual host All Cluster rabbit@DESKTOP-J208P11 User guest Log out

Overview Connections Channels Exchanges Queues Admin

Overview

Totals

Queued messages last minute ?

Currently idle

Message rates last minute ?

Currently idle

Global counts ?

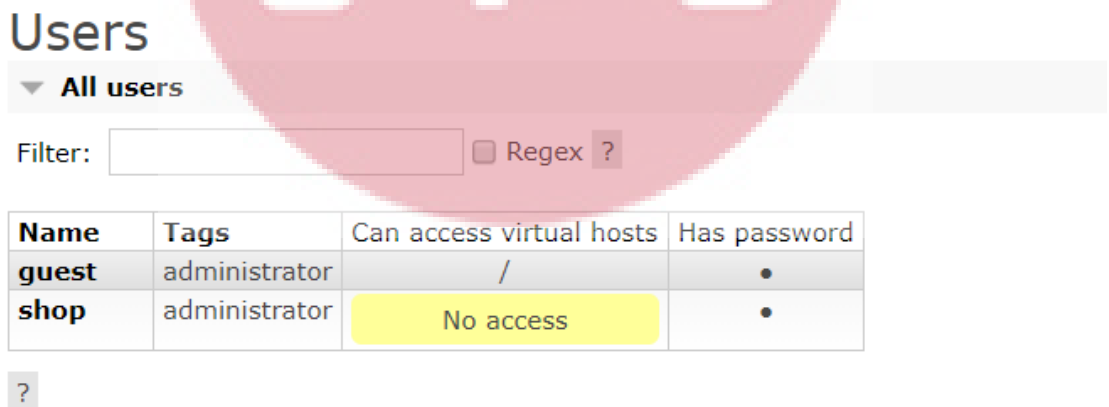
Connections: 0 Channels: 0 Exchanges: 7 Queues: 0 Consumers: 0

添加用户，默认用户 `guest` 角色为管理员，一般开发环境下会创建新的用户并对权限进行分配。

添加 `lego` 用户并对权限进行分配



用户添加完毕，用户列表显示用户状态是 `No access` ,代表用户未进行权限分配，不能进行任何操作，这里创建用户分配权限可以类比数据库中创建用户并分配权限操作。



分配权限

创建 `virtual hosts` 可以类比创建数据库，分配用户操作权限



[Overview](#) [Connections](#) [Channels](#) [Exchanges](#) [Queues](#) [Admin](#)

Virtual Hosts

▼ All virtual hosts

Filter: ☐ Regex ?

Overview			Messages			Network		Message rates		+/-
Name	Users ?	State	Ready	Unacked	Total	From client	To client	publish	deliver / get	
/	guest	■ running	0	0	0					

▼ Add a new virtual host

Name: *

[Add virtual host](#)

创建完成以后可以看到默认分配了 `guest` 用户

Virtual Hosts

▼ All virtual hosts

Filter: ☐ Regex ?

Overview			Messages			Network		Message rates		+/-
Name	Users ?	State	Ready	Unacked	Total	From client	To client	publish	deliver / get	
/	guest	■ running	0	0	0					
/shop	guest	■ running	NaN	NaN	NaN					

点击 `/shop` 进入权限分配页面，添加用户

Virtual Hosts

▼ All virtual hosts

Filter: ☐ Regex ?

Overview			Messages			Network		Message rates		+/-
Name	Users ?	State	Ready	Unacked	Total	From client	To client	publish	deliver / get	
/shop	guest	■ running	0	0	0					
	guest	■ running	NaN	NaN	NaN					

添加 `shop` 用户



▼ Permissions

Current permissions

User	Configure regexp	Write regexp	Read regexp	
guest	.*	.*	.*	Clear

Set permission

User

Configure regexp:

Write regexp:

Read regexp:

Set permission

添加成功如下，Clear代表清除用户

▼ Permissions

Current permissions

User	Configure regexp	Write regexp	Read regexp	
guest	.*	.*	.*	Clear
shop	.*	.*	.*	Clear

Set permission

User

Configure regexp:

Write regexp:

Read regexp:

Set permission

Virtual Hosts

▼ All virtual hosts

Filter: ☐ Regex ?

Overview			Messages			Network		Message rates		
Name	Users ?	State	Ready	Unacked	Total	From client	To client	publish	deliver / get	
/	guest	running	0	0	0					
/shop	guest, shop	running	NaN	NaN	NaN					

返回Users，现在shop用户只有可以操作/shop的权限，配置/操作权限



Users

▼ All users

Filter: ☐ Regex ?

Name	Tags	Can access virtual hosts	Has password
guest	administrator	/, /shop	•
shop	administrator	/shop	•

?

点击ego进入权限分配页面

Users

▼ All users

Filter: ☐ Regex ?

Name	Tags	Can access virtual hosts	Has password
guest	administrator	/, /shop	•
shop	administrator	/shop	•

?

添加/操作权限

User: shop

▼ Overview

Tags administrator

Can log in with password •

▼ Permissions

Current permissions

Virtual host	Configure regexp	Write regexp	Read regexp	
/shop	.*	.*	.*	Clear

Set permission

Virtual Host: / ▼

Configure regexp: .*

Write regexp: .*

Read regexp: .*

Set permission

添加成功如下，Clear代表清除用户



user: shop

Overview

Tags

administrator

Can log in with password

•

Permissions

Current permissions

Virtual host	Configure regexp	Write regexp	Read regexp	
/	.*	.*	.*	Clear
/shop	.*	.*	.*	Clear

Set permission

Virtual Host: /

Configure regexp: .*

Write regexp: .*

Read regexp: .*

Set permission

重新登录

现在我们就可以使用shop用户登录RabbitMQ

Refreshed 2020-02-20 22:22:16 | Refresh every 5 seconds

Virtual host: All

Cluster: rabbit@DESKTOP-A1913US

User: shop | Log out

User: shop

Overview

Tags: administrator

Can log in with password: •

Permissions

Current permissions

Virtual host	Configure regexp	Write regexp	Read regexp	
/	.*	.*	.*	Clear
/shop	.*	.*	.*	Clear

Set permission

Virtual Host: /

Configure regexp: .*

Write regexp: .*

Read regexp: .*

Set permission

角色，权限讲解

RabbitMQ角色分类

- none:不能访问 management plugin（管理插件）
- impersonator:演员？？？
- 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。



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

polymaker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

RabbitMQ权限控制

默认virtual host: "/"

默认用户: guest

guest具有 "/" 上的全部权限, 仅能有localhost访问RabbitMQ包括Plugin, 建议删除或更改密码。

可通过将配置文件中 `loopback_users` 置空来取消其本地访问的限制: `[{rabbit, [{loopback_users, []}]}`]

用户仅能对其所能访问的virtual hosts中的资源进行操作。这里的资源指的是virtual hosts中的exchanges、queues等, 操作包括对资源进行配置、写、读。配置权限可创建、删除资源并修改资源的行为, 写权限可向资源发送消息, 读权限从资源获取消息。

比如:

- exchange和queue的declare与delete分别需要exchange和queue上的配置权限
- exchange的bind与unbind需要exchange的读写权限
- queue的bind与unbind需要queue写权限exchange的读权限
- 发消息(publish)需exchange的写权限
- 获取或清除(get、consume、purge)消息需queue的读权限
- 对何种资源具有配置、写、读的权限通过正则表达式来匹配, 具体命令如下: `set_permissions [-p <vhostpath>] <user> <conf> <write> <read>`, 其中, `<conf> <write> <read>` 的位置分别用正则表达式来匹配特定的资源, 如 `'^(amq\.gen\.*|amq\.default)$'` 可以匹配server生成的和默认的exchange, `'^$'` 不匹配任何资源

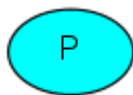
要注意的是RabbitMQ会缓存每个connection或channel的权限验证结果、因此权限发生变化后需要重连才能生效。

RabbitMQ专业术语

官网地址: <http://www.rabbitmq.com/getstarted.html>

Producing

Producing意思不仅仅是发送消息。发送消息的程序叫做producer生产者。



Queue

Queue是一个消息盒子的名称。它存活在 RabbitMQ 里。虽然消息流经 RabbitMQ 和你的应用程序，但是他们只能在 Queue 里才能被保存。Queue 没有任何边界的限制，你想存多少消息都可以，它本质上是一个无限的缓存。许多生产者都可以向一个 Queue 里发送消息，许多消费者都可以从一个 Queue 里接收消息。

queue_name



Consuming

Consuming 的意思和接收类似。等待接收消息的程序叫做消费者。



注意：生产者，消费者和代理不一定非要在同一台机器上。

ConnectionFactory、Connection、Channel

`ConnectionFactory`、`Connection`、`Channel` 都是RabbitMQ对外提供的API中最基本的对象。

`Connection` 是RabbitMQ的 `socket` 连接，它封装了 `socket` 协议相关部分逻辑。

`ConnectionFactory` 为`Connection`的制造工厂。

`Channel` 是我们与RabbitMQ打交道的最重要的一个接口，我们大部分的业务操作是在Channel这个接口中完成的，包括定义 `Queue`、定义 `Exchange`、绑定 `Queue` 与 `Exchange`、发布消息等。

Message acknowledgment

在实际应用中，可能会发生消费者收到 `Queue` 中的消息，但没有处理完成就宕机（或出现其他意外）的情况，这种情况下就可能会导致消息丢失。为了避免这种情况发生，我们可以要求消费者在消费完消息后发送一个回执给RabbitMQ，RabbitMQ收到消息回执（`Message acknowledgment`）后才将该消息从 `Queue` 中移除；如果RabbitMQ没有收到回执并检测到消费者的RabbitMQ连接断开，则RabbitMQ会将该消息发送给其他消费者（如果存在多个消费者）进行处理。这里不存在timeout概念，一个消费者处理消息时间再长也不会导致该消息被发送给其他消费者，除非它的RabbitMQ连接断开。

这里会产生另外一个问题，如果我们的开发人员在处理完业务逻辑后，忘记发送回执给RabbitMQ，这将会导致严重的bug——`Queue` 中堆积的消息会越来越多；消费者重启后会重复消费这些消息并重复执行业务逻辑...

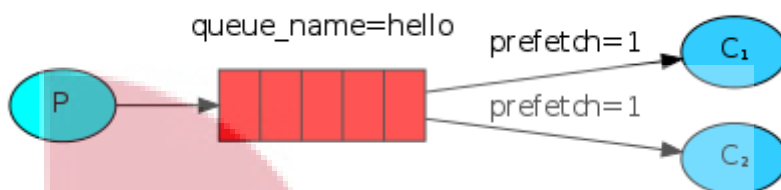
Message durability



如果我们希望即使在RabbitMQ服务重启的情况下，也不会丢失消息，我们可以将Queue与Message都设置为可持久化的（durable），这样可以保证绝大部分情况下我们的RabbitMQ消息不会丢失。但依然解决不了小概率丢失事件的发生（比如RabbitMQ服务器已经接收到生产者的消息，但还没来得及持久化该消息时RabbitMQ服务器就断电了），如果我们需要对这种小概率事件也要管理起来，那么我们要用到事务。

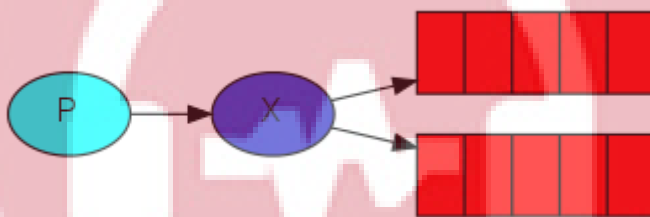
Prefetch count

前面我们讲到如果有多个消费者同时订阅同一个 Queue 中的消息，Queue 中的消息会被平摊给多个消费者。这时如果每个消息的处理时间不同，就有可能导致某些消费者一直在忙，而另外一些消费者很快就处理完手头工作并一直空闲的情况。我们可以通过设置 prefetchCount 来限制 Queue 每次发送给每个消费者的消息数，比如我们设置prefetchCount=1，则 queue 每次给每个消费者发送一条消息；消费者处理完这条消息后 queue 会再给该消费者发送一条消息。



Exchange

生产者将消息投递到 Queue 中，实际上这在RabbitMQ中这种事情永远都不会发生。实际的情况是，生产者将消息发送到Exchange（交换器，下图中的X），由Exchange将消息路由到一个或多个 Queue 中（或者丢弃）。



Exchange是按照什么逻辑将消息路由到 Queue 的？这个将在Binding一节介绍。

RabbitMQ中的Exchange有四种类型，不同的类型有着不同的路由策略，这将在Exchange Types一节介绍。

routing key

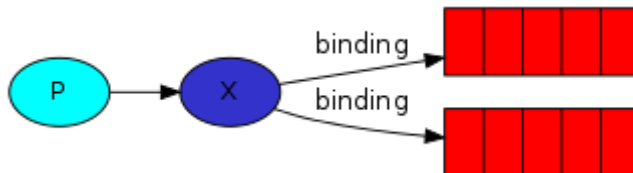
生产者在将消息发送给Exchange的时候，一般会指定一个 routing key，来指定这个消息的路由规则，而这个 routing key 需要与 Exchange Type 及 binding key 联合使用才能最终生效。

在 Exchange Type 与 binding key 固定的情况下（在正常使用时一般这些内容都是固定配置好的），我们的生产者就可以在发送消息给Exchange时，通过指定 routing key 来决定消息流向哪里。

RabbitMQ为 routing key 设定的长度限制为255 bytes。

Binding

RabbitMQ中通过Binding将Exchange与Queue关联起来，这样RabbitMQ就知道如何正确地将消息路由到指定的Queue了。



Binding key

在绑定 (Binding) Exchange 与 Queue 的同时，一般会指定一个 `binding key`；消费者将消息发送给 Exchange 时，一般会指定一个 `routing key`；当 `binding key` 与 `routing key` 相匹配时，消息将会被路由到对应的 Queue 中。这个将在 Exchange Types 章节会列举实际的例子加以说明。

在绑定多个 Queue 到同一个 Exchange 的时候，这些 Binding 允许使用相同的 `binding key`。

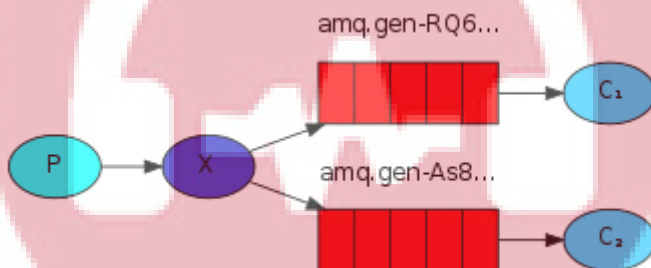
`binding key` 并不是在所有情况下都生效，它依赖于 Exchange Type，比如 `fanout` 类型的 Exchange 就会无视 `binding key`，而是将消息路由到所有绑定到该 Exchange 的 Queue。

Exchange Types

RabbitMQ 常用的 Exchange Type 有 `fanout`、`direct`、`topic`、`headers` 这四种（AMQP 规范里还提到两种 Exchange Type，分别为 `system` 与 自定义，这里不予以描述），下面分别进行介绍。

fanout

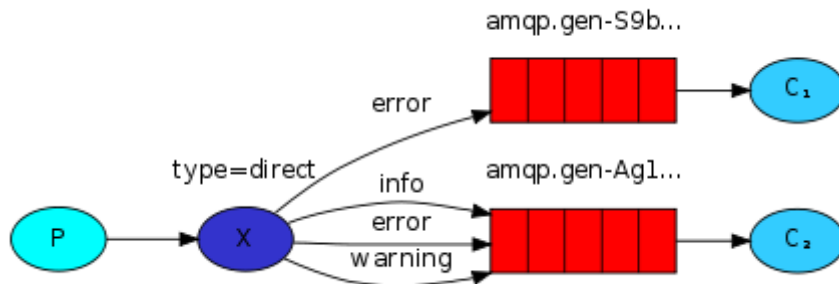
`fanout` 类型的 Exchange 路由规则非常简单，它会把所有发送到该 Exchange 的消息路由到所有与它绑定的 Queue 中。



上图中，生产者 (P) 发送到 Exchange (X) 的所有消息都会路由到图中的两个 Queue，并最终被两个消费者 (C1 与 C2) 消费。

direct

`direct` 类型的 Exchange 路由规则也很简单，它会把消息路由到那些 `binding key` 与 `routing key` 完全匹配的 Queue 中。





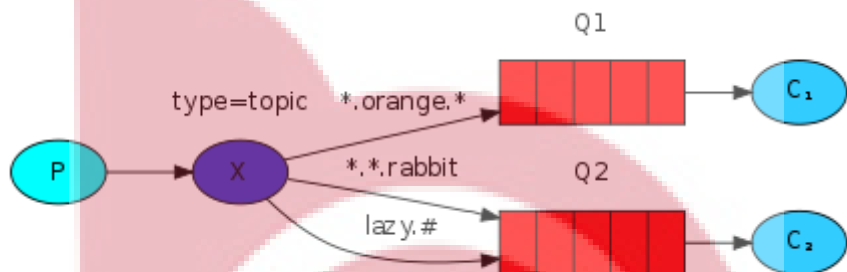
以上图的配置为例，我们以 `routingKey="error"` 发送消息到Exchange，则消息会路由到

Queue1 (amqp.gen-S9b..., 这是由RabbitMQ自动生成的Queue名称) 和Queue2 (amqp.gen-Agl...); 如果我们以 `routingKey="info"` 或 `routingKey="warning"` 来发送消息，则消息只会路由到Queue2。如果我们以其他routingKey发送消息，则消息不会路由到这两个Queue中。

topic

前面讲到direct类型的Exchange路由规则是完全匹配 `binding key` 与 `routing key`，但这种严格的匹配方式在很多情况下不能满足实际业务需求。topic类型的Exchange在匹配规则上进行了扩展，它与direct类型的Exchange相似，也是将消息路由到 `binding key` 与 `routing key` 相匹配的Queue中，但这里的匹配规则有些不同，它约定：

- routing key为一个句点号 . 分隔的字符串（我们将被句点号 . 分隔开的每一段独立的字符串称为一个单词），如“stock.usd.nyse”、“nyse.vmw”、“quick.orange.rabbit”
- binding key与routing key一样也是句点号 . 分隔的字符串
- binding key中可以存在两种特殊字符 * 与 #，用于做模糊匹配，其中 * 用于匹配一个单词，# 用于匹配多个单词（可以是零个）



以上图中的配置为例，`routingKey="quick.orange.rabbit"` 的消息会同时路由到Q1与Q2，`routingKey="lazy.orange.fox"` 的消息会路由到Q1，`routingKey="lazy.brown.fox"` 的消息会路由到Q2，`routingKey="lazy.pink.rabbit"` 的消息会路由到Q2（只会投递给Q2一次，虽然这个routingKey与Q2的两个bindingKey都匹配）；`routingKey="quick.brown.fox"`、`routingKey="orange"`、`routingKey="quick.orange.male.rabbit"` 的消息将会被丢弃，因为它们没有匹配任何bindingKey。

headers

headers类型的Exchange不依赖于 `routing key` 与 `binding key` 的匹配规则来路由消息，而是根据发送的消息内容中的headers属性进行匹配。

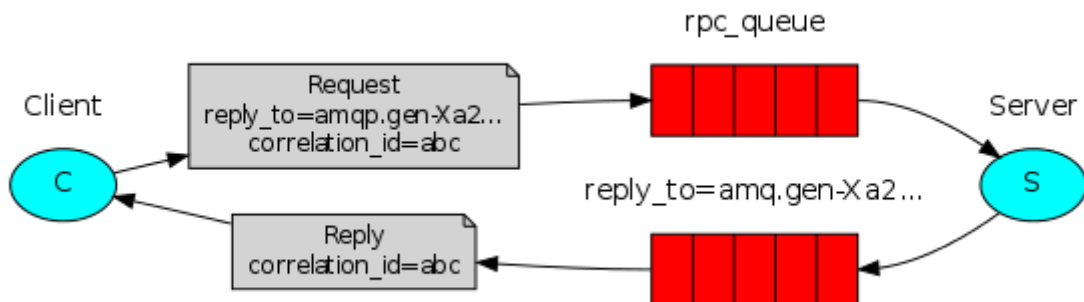
在绑定Queue与Exchange时指定一组键值对；当消息发送到Exchange时，RabbitMQ会取到该消息的headers（也是一个键值对的形式），对比其中的键值对是否完全匹配Queue与Exchange绑定时指定的键值对；如果完全匹配则消息会路由到该Queue，否则不会路由到该Queue。

该类型的Exchange目前用的不多（不过也应该很有用武之地），因此不做重点介绍。

RPC

MQ本身是基于异步的消息处理，前面的示例中所有的生产者（P）将消息发送到RabbitMQ后不会知道消费者（C）处理成功或者失败（甚至连有没有消费者来处理这条消息都不知道）。

但实际的应用场景中，我们很可能需要一些同步处理，需要同步等待服务端将我的消息处理完成后再进行下一步处理。这相当于RPC（Remote Procedure Call，远程过程调用）。在RabbitMQ中也支持RPC。



RabbitMQ中实现RPC的机制是：

- 客户端发送请求（消息）时，在消息的属性（`MessageProperties`，在AMQP协议中定义了14种 properties，这些属性会随着消息一起发送）中设置两个值 `replyTo`（一个Queue名称，用于告诉服务器处理完成后将通知我的消息发送到这个Queue中）和 `correlationId`（此次请求的标识号，服务器处理完成后需要将此属性返还，客户端将根据这个id了解哪条请求被成功执行了或执行失败）
- 服务器端收到消息并处理
- 服务器端处理完消息后，将生成一条应答消息到 `replyTo` 指定的Queue，同时带上 `correlationId` 属性
- 客户端之前已订阅 `replyTo` 指定的Queue，从中收到服务器的应答消息后，根据其中的 `correlationId` 属性分析哪条请求被执行了，根据执行结果进行后续业务处理

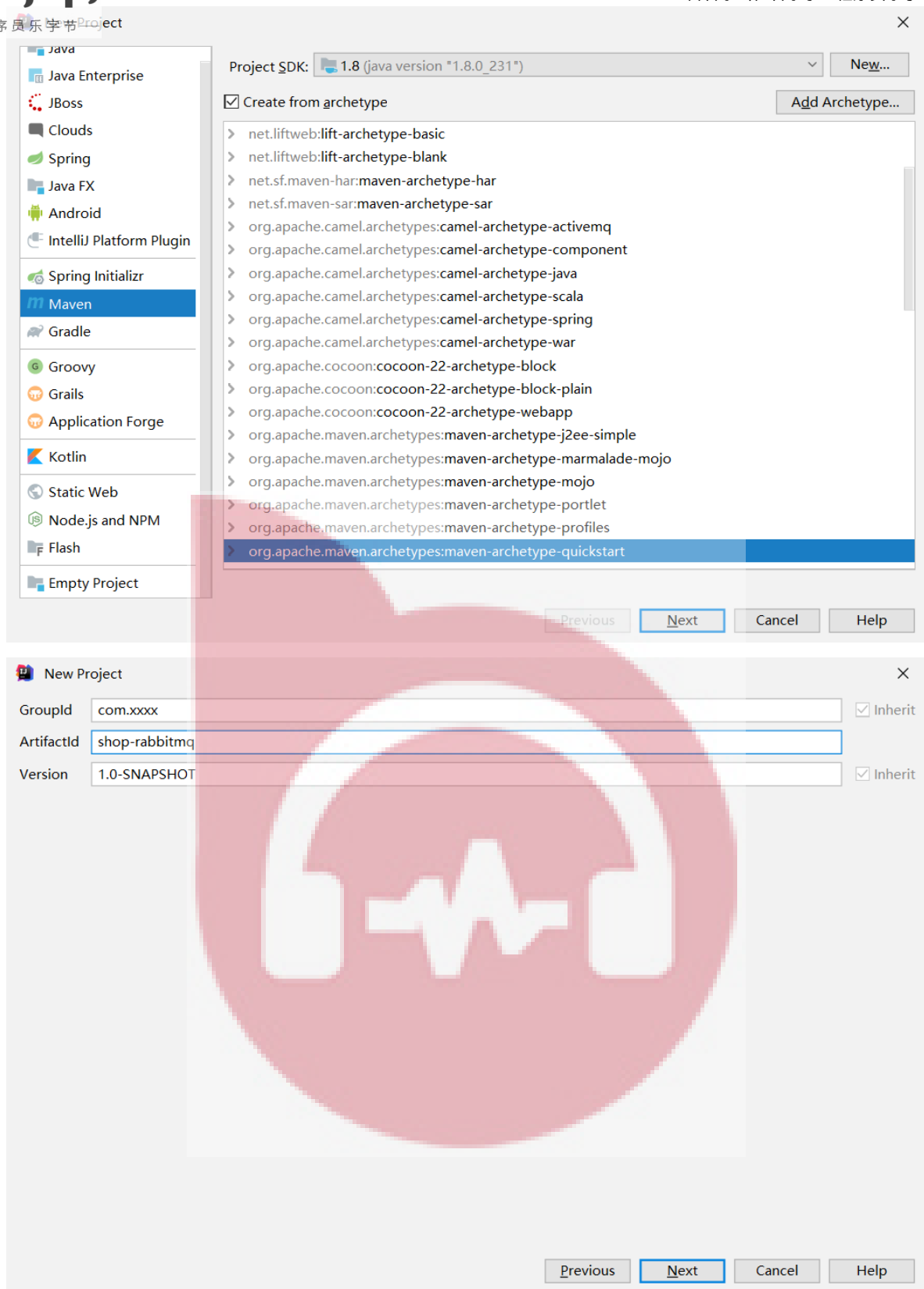
简单模式队列

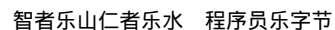
在这部分的使用指南中，我们要用Java写两个程序；一个是生产者，他发送一个消息，另一个是消费者，它接收消息，并且把消息打印出来。我们将会忽略一些Java API的细节，而是将注意力主要放在我们将来要做的这件事上，这件事就是发送一个 "Hello World" 消息。


在下面的图中，"P" 代表生产者，而 "C" 代表消费者。中间的就是一个 Queue，一个消息缓存区。



创建项目





 New Project ✕

Project name:

shop-rabbitmq

Project location:

D:\workspace\demo\shop-rabbitmq

...

▶ More Settings

Previous

Finish

Cancel

Help

第 21 / 76页
上海市浦东新区汇通南园文化创意园



RabbitMQ Java Client » 5.6.0

The RabbitMQ Java client library allows Java applications to interface with RabbitMQ.

License	Apache 2.0 GPL 2.0 MPL 1.1
Categories	Message Queue Clients
Organization	Pivotal Software, Inc.
HomePage	http://www.rabbitmq.com
Date	(Jan 25, 2019)
Files	jar (586 KB) View All
Repositories	Central
Used By	391 artifacts

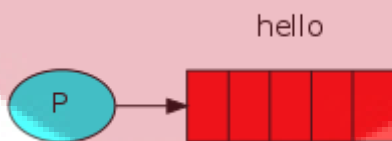
Maven Gradle SBT Ivy Grape Leiningen Buildr

```
<!-- https://mvnrepository.com/artifact/com.rabbitmq/amqp-client -->
<dependency>
  <groupId>com.rabbitmq</groupId>
  <artifactId>amqp-client</artifactId>
  <version>5.6.0</version>
</dependency>
```

```
<!-- rabbitmq依赖 -->
<dependency>
  <groupId>com.rabbitmq</groupId>
  <artifactId>amqp-client</artifactId>
  <version>5.6.0</version>
</dependency>
```

Sending

我们把消息发送者叫 Send，消息接收者叫 Recv。消息发送者连接 RabbitMQ，发送一个消息，然后退出。



Send.java

```
package com.xxxx.simple.send;

import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;

import java.io.IOException;
import java.util.concurrent.TimeoutException;

/**
 * 简单模式队列-消息发送者
 */
public class Send {
```



```
private final static String QUEUE_NAME = "hello";

public static void main(String[] args) {
    // 创建连接工厂
    ConnectionFactory factory = new ConnectionFactory();
    factory.setHost("127.0.0.1");
    factory.setPort(5672);
    factory.setUsername("shop");
    factory.setPassword("shop");
    factory.setVirtualHost("/shop");

    Connection connection = null;
    Channel channel = null;
    try {
        // 通过工厂创建连接
        connection = factory.newConnection();
        // 获取通道
        channel = connection.createChannel();
        /**
         * 声明队列
         * 第一个参数queue: 队列名称
         * 第二个参数durable: 是否持久化
         * 第三个参数Exclusive: 排他队列，如果一个队列被声明为排他队列，该队列仅对首次
         * 声明它的连接可见，并在连接断开时自动删除。
         * 这里需要注意三点：
         * 1. 排他队列是基于连接可见的，同一连接的不同通道是可以同时访问同一个
         * 连接创建的排他队列的。
         * 2. "首次"，如果一个连接已经声明了一个排他队列，其他连接是不允许建
         * 立同名的排他队列的，这个与普通队列不同。
         * 3. 即使该队列是持久化的，一旦连接关闭或者客户端退出，该排他队列都会
         * 被自动删除的。
         * 这种队列适用于只限于一个客户端发送读取消息的应用场景。
         * 第四个参数Auto-delete: 自动删除，如果该队列没有任何订阅的消费者的话，该队列
         * 会被自动删除。
         * 这种队列适用于临时队列。
         */
        channel.queueDeclare(QUEUE_NAME, false, false, false, null);
        // 创建消息
        String message = "Hello world!";
        // 将产生的消息放入队列
        channel.basicPublish("", QUEUE_NAME, null, message.getBytes("UTF-
8"));

        System.out.println(" [x] Sent '" + message + "'");
    } catch (IOException e) {
        e.printStackTrace();
    } catch (TimeoutException e) {
        e.printStackTrace();
    } finally {
        try {
            // 关闭通道
            if (null != channel && channel.isOpen())
                channel.close();
            // 关闭连接
            if (null != connection && connection.isOpen())
                connection.close();
        } catch (TimeoutException e) {
            e.printStackTrace();
        }
    }
}
```



```
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
}  
}  
}
```

消息发送成功以后，通过RabbitMQ管理界面可以看到队列的相关信息

Queues

▼ All queues (1)

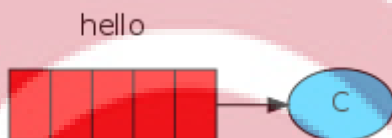
Pagination

Page 1 of 1 - Filter: ☐ Regex ?

Overview				Messages			Message rates				+/-
Virtual host	Name	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack		
/shop	hello		idle	1	0	1	0.00/s				

Receiving

消息的发送者只是发送一个消息，我们的接收者需要不断的监听消息，并把它们打印出来。



Recv.java

```
package com.xxxx.simple.recv;  
  
import com.rabbitmq.client.*;  
  
import java.io.IOException;  
import java.util.concurrent.TimeoutException;  
  
/**  
 * 简单模式队列-消息接收者  
 */  
public class Recv {  
  
    // 队列名称  
    private final static String QUEUE_NAME = "hello";  
  
    public static void main(String[] args) {  
        // 创建连接工厂  
        ConnectionFactory factory = new ConnectionFactory();  
        factory.setHost("127.0.0.1");  
        factory.setPort(5672);  
        factory.setUsername("shop");  
        factory.setPassword("shop");  
        factory.setVirtualHost("/shop");  
  
        try {  
            // 通过工厂创建连接
```




```
Connection connection = factory.newConnection();
// 获取通道
Channel channel = connection.createChannel();
// 指定队列
channel.queueDeclare(QUEUE_NAME, false, false, false, null);
System.out.println("[*] Waiting for messages. To exit press
CTRL+C");

// -----之前旧版本的写法-----begin-----
/*
// 获取消息
Consumer consumer = new DefaultConsumer(channel) {
    @Override
    public void handleDelivery(String consumerTag, Envelope
envelope,

                                AMQP.BasicProperties properties,
                                byte[] body) throws IOException {

        // 获取消息并在控制台打印
        String message = new String(body, "utf-8");
        System.out.println("[x] Received '" + message + "'");
    }
};
// 监听队列
channel.basicConsume(QUEUE_NAME, true, consumer);
*/
// -----之前旧版本的写法-----end-----

// 获取消息
DeliverCallback deliverCallback = (consumerTag, delivery) -> {
    String message = new String(delivery.getBody(), "UTF-8");
    System.out.println("[x] Received '" + message + "'");
};
// 监听队列
channel.basicConsume(QUEUE_NAME, true, deliverCallback, consumerTag
-> {
    });
} catch (IOException e) {
    e.printStackTrace();
} catch (TimeoutException e) {
    e.printStackTrace();
}
}
}
```

消息接收成功以后，通过RabbitMQ管理界面可以看到队列的相关信息

Queues

▼ All queues (1)

Pagination

Page 1 of 1 - Filter: ☐ Regex ?

Overview				Messages			Message rates			+/-
Virtual host	Name	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack	
/shop	hello		idle	0	0	0	0.00/s	0.00/s	0.00/s	

测试



```
Run: Recv x Send x
D:\Java\jdk1.8.0_202\bin\java.exe ...
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
[x] Sent 'Hello World!'
Process finished with exit code 0
```

运行Recv.java

```
Run: Recv x Send x
[x] Received 'Hello World!'
```

总结

问题：如果任务量很大，消息得不到及时的消费会造成队列积压，问题非常严重，比如内存溢出，消息丢失等。

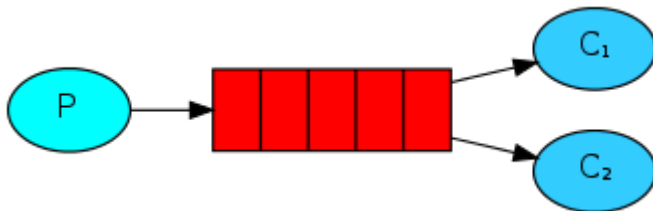
解决：配置多个消费者消费消息。

总结：简单队列-处理消息效率不高，吞吐量较低，不适合生成环境

Work queues-工作模式队列

工作模式队列-消息轮询分发(Round-robin)

通过Helloworld工程我们已经能够构建一个简单的消息队列的基本项目，项目中存在几个角色:生产者、消费者、队列，而对于我们真实的开发中，对于消息的消费者通过是有多个的，比如在实现用户注册功能时，用户注册成功，会给响对应用户发送邮件，同时给用户发送手机短信，告诉用户已成功注册网站或者app 应用，这种功能在大部分项目开发中都比较常见，而对于helloworld 的应用中虽然能够对消息进行消费，但是有很大问题:消息消费者只有一个，当消息量非常大时，单个消费者处理消息就会变得很慢，同时给节点页带来很大压力，导致消息堆积越来越多。对于这种情况，RabbitMQ 提供了工作队列模式，通过工作队列提供做个消费者，对MQ产生的消息进行消费，提高MQ消息的吞吐率，降低消息的处理时间。处理模型图如下



Sending

Send.java

```
package com.xxxx.work.roundRobin.send;
```



```
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;

import java.io.IOException;
import java.util.concurrent.TimeoutException;

/**
 * 工作模式队列-轮询分发-消息发送者
 */
public class Send {

    // 队列名称
    private final static String QUEUE_NAME = "work_roundRobin";

    public static void main(String[] args) {
        // 创建连接工厂
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("127.0.0.1");
        factory.setPort(5672);
        factory.setUsername("shop");
        factory.setPassword("shop");
        factory.setVirtualHost("/shop");

        Connection connection = null;
        Channel channel = null;
        try {
            // 通过工厂创建连接
            connection = factory.newConnection();
            // 获取通道
            channel = connection.createChannel();
            // 声明队列
            channel.queueDeclare(QUEUE_NAME, false, false, false, null);
            // 创建消息
            for (int i = 1; i <= 20; i++) {
                String message = "Hello world! ----- " + i;
                // 将产生的消息放入队列
                channel.basicPublish("", QUEUE_NAME, null,
message.getBytes("UTF-8"));
                System.out.println(" [x] Sent '" + message + "'");
            }
        } catch (IOException e) {
            e.printStackTrace();
        } catch (TimeoutException e) {
            e.printStackTrace();
        } finally {
            try {
                // 关闭通道
                if (null != channel && channel.isOpen())
                    channel.close();
                // 关闭连接
                if (null != connection && connection.isOpen())
                    connection.close();
            } catch (TimeoutException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```



```
}  
}  
}  
}
```

Receiving

Recv01.java

```
package com.xxxx.work.roundRobin.recv;  
  
import com.rabbitmq.client.Channel;  
import com.rabbitmq.client.Connection;  
import com.rabbitmq.client.ConnectionFactory;  
import com.rabbitmq.client.DeliverCallback;  
  
import java.io.IOException;  
import java.util.concurrent.TimeoutException;  
  
/**  
 * 工作模式队列-轮询分发-消息接收者  
 */  
public class Recv01 {  
  
    // 队列名称  
    private final static String QUEUE_NAME = "work_roundRobin";  
  
    public static void main(String[] args) {  
        // 创建连接工厂  
        ConnectionFactory factory = new ConnectionFactory();  
        factory.setHost("127.0.0.1");  
        factory.setPort(5672);  
        factory.setUsername("shop");  
        factory.setPassword("shop");  
        factory.setVirtualHost("/shop");  
  
        try {  
            // 通过工厂创建连接  
            Connection connection = factory.newConnection();  
            // 获取通道  
            Channel channel = connection.createChannel();  
            // 指定队列  
            channel.queueDeclare(QUEUE_NAME, false, false, false, null);  
            System.out.println(" [*] waiting for messages. To exit press  
CTRL+C");  
            // 获取消息  
            DeliverCallback deliverCallback = (consumerTag, delivery) -> {  
                String message = new String(delivery.getBody(), "UTF-8");  
                System.out.println(" [x] Received01 '" + message + "'");  
                // 模拟程序执行所耗时间  
                try {  
                    Thread.sleep(1000);  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
            };  
        }  
    }  
};
```



```
// 监听队列
channel.basicConsume(QueueName, true, deliverCallback, consumerTag
-> {
    });
} catch (IOException e) {
    e.printStackTrace();
} catch (TimeoutException e) {
    e.printStackTrace();
}
}
```

Recv02.java

```
package com.xxxx.work.roundRobin.recv;

import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.DeliverCallback;

import java.io.IOException;
import java.util.concurrent.TimeoutException;

/**
 * 工作模式队列-轮询分发-消息接收者
 */
public class Recv02 {

    // 队列名称
    private final static String QUEUE_NAME = "work_roundRobin";

    public static void main(String[] args) {
        // 创建连接工厂
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("127.0.0.1");
        factory.setPort(5672);
        factory.setUsername("shop");
        factory.setPassword("shop");
        factory.setVirtualHost("/shop");

        try {
            // 通过工厂创建连接
            Connection connection = factory.newConnection();
            // 获取通道
            Channel channel = connection.createChannel();
            // 指定队列
            channel.queueDeclare(QUEUE_NAME, false, false, false, null);
            System.out.println("[*] waiting for messages. To exit press CTRL+C");

            // 获取消息
            DeliverCallback deliverCallback = (consumerTag, delivery) -> {
                String message = new String(delivery.getBody(), "UTF-8");
                System.out.println("[x] Received02 '" + message + "'");
                // 模拟程序执行所耗时间
                try {

```



```
        Thread.sleep(2000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
};
// 监听队列
channel.basicConsume(QueueName, true, deliverCallback, consumerTag
-> {
    });
} catch (IOException e) {
    e.printStackTrace();
} catch (TimeoutException e) {
    e.printStackTrace();
}
}
```

测试

运行Send

运行Recv

```
Run: Recv01 x Recv02 x Send (1) x
[x] Received01 'Hello World! ----- 1'
[x] Received01 'Hello World! ----- 3'
[x] Received01 'Hello World! ----- 5'
[x] Received01 'Hello World! ----- 7'
[x] Received01 'Hello World! ----- 9'
[x] Received01 'Hello World! ----- 11'
[x] Received01 'Hello World! ----- 13'
[x] Received01 'Hello World! ----- 15'
[x] Received01 'Hello World! ----- 17'
[x] Received01 'Hello World! ----- 19'
```

```
Run: Recv01 x Recv02 x Send (1) x
[x] Received02 'Hello World! ----- 2'
[x] Received02 'Hello World! ----- 4'
[x] Received02 'Hello World! ----- 6'
[x] Received02 'Hello World! ----- 8'
[x] Received02 'Hello World! ----- 10'
[x] Received02 'Hello World! ----- 12'
[x] Received02 'Hello World! ----- 14'
[x] Received02 'Hello World! ----- 16'
[x] Received02 'Hello World! ----- 18'
[x] Received02 'Hello World! ----- 20'
```

总结

从结果可以看出消息被平均分配到两个消费方，来对消息进行处理，提高了消息处理效率，创建多个消费者来对消息进行处理。这里RabbitMQ采用轮询来对消息进行分发时保证了消息被平均分配到每个消费方，但是引入新的问题:真正的生产环境下，对于消息的处理基本不会像我们现在看到的这样，每个消费方处理的消息数量是平均分配的，比如因为网络原因，机器cpu,内存等硬件问题，消费方处理消息时同类消息不同机器进行处理时消耗时间也是不一样的，比如1号消费者消费1条消息时1秒，2号消费者消费1条消息是5秒，对于1号消费者比2号消费者处理消息快，那么在分配消息时就应该让1号消费者多收到消息进行处理，也即是我们通常所说的“能者多劳”,同样Rabbitmq对于这种消息分配模式提供了支持。

问题：任务量很大，消息虽然得到了及时的消费，单位时间内消息处理速度加快，提高了吞吐量，可是不同消费者处理消息的时间不同，导致部分消费者的资源被浪费。

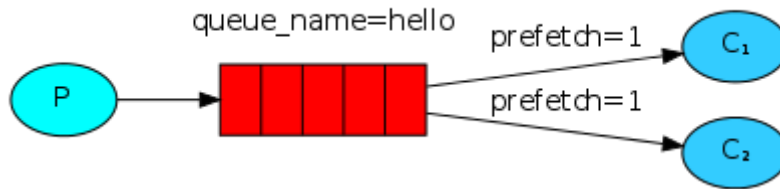
解决：采用消息公平分发。

总结：工作队列-消息轮询分发-消费者收到的消息数量平均分配，单位时间内消息处理速度加快，提高了吞吐量。

工作模式队列-消息公平分发(fair dispatch)

在案例01中对于消息分发采用的是默认轮询分发，消息应答采用的自动应答模式，这是因为当消息进入队列，RabbitMQ就会分派消息。它不看消费者为应答的数目，只是盲目的将第n条消息发给第n个消费者。

为了解决这个问题，我们使用 `basicQos(prefetchCount = 1)` 方法，来限制RabbitMQ只发不超过1条的消息给同一个消费者。当消息处理完毕后，有了反馈，才会进行第二次发送。执行模型图如下：



Sending

Send.java

```
package com.xxxx.work.fair.send;

import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;

import java.io.IOException;
import java.util.concurrent.TimeoutException;

/**
 * 工作模式队列-公平分发-消息发送者
 */
public class Send {

    // 队列名称
    private final static String QUEUE_NAME = "work_fair";

    public static void main(String[] args) {
        // 创建连接工厂
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("127.0.0.1");
        factory.setPort(5672);
        factory.setUsername("shop");
        factory.setPassword("shop");
        factory.setVirtualHost("/shop");

        Connection connection = null;
        Channel channel = null;
        try {
            // 通过工厂创建连接
            connection = factory.newConnection();
            // 获取通道
            channel = connection.createChannel();
            // 声明队列
            channel.queueDeclare(QUEUE_NAME, false, false, false, null);
            // 创建消息
            for (int i = 1; i <= 20; i++) {
                String message = "Hello world! ----- " + i;
                // 将产生的消息放入队列
                channel.basicPublish("", QUEUE_NAME, null,
message.getBytes("UTF-8"));
                System.out.println(" [x] Sent '" + message + "'");
            }
        } catch (IOException e) {
            e.printStackTrace();
        } catch (TimeoutException e) {
        }
    }
}
```




```
e.printStackTrace();
} finally {
    try {
        // 关闭通道
        if (null != channel && channel.isOpen())
            channel.close();
        // 关闭连接
        if (null != connection && connection.isOpen())
            connection.close();
    } catch (TimeoutException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}
}

package com.xxxx.work.fair.recv;

import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.DeliverCallback;

import java.io.IOException;
import java.util.concurrent.TimeoutException;

/**
 * 工作模式队列-公平分发-消息接收者
 */
public class Recv01 {

    // 队列名称
    private final static String QUEUE_NAME = "work_fair";

    public static void main(String[] args) {
        // 创建连接工厂
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("127.0.0.1");
        factory.setPort(5672);
        factory.setUsername("shop");
        factory.setPassword("shop");
        factory.setVirtualHost("/shop");

        try {
            // 通过工厂创建连接
            final Connection connection = factory.newConnection();
            // 获取通道
            final Channel channel = connection.createChannel();
            // 指定队列
            channel.queueDeclare(QUEUE_NAME, false, false, false, null);
```

```

/*
    限制RabbitMQ只发不超过1条的消息给同一个消费者。
    当消息处理完毕后，有了反馈，才会进行第二次发送。
*/
int prefetchCount = 1;
channel.basicQos(prefetchCount);

System.out.println(" [*] waiting for messages. To exit press
CTRL+C");

// 获取消息
DeliverCallback deliverCallback = (consumerTag, delivery) -> {
    String message = new String(delivery.getBody(), "UTF-8");
    System.out.println(" [x] Received01 '" + message + "'");
    // 模拟程序执行所耗时间
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    // 手动回执消息
    channel.basicAck(delivery.getEnvelope().getDeliveryTag(),
false);
};
// 监听队列
/*
    autoAck = true代表自动确认消息
    autoAck = false代表手动确认消息
*/
boolean autoAck = false;
channel.basicConsume(QUEUE_NAME, autoAck, deliverCallback,
consumerTag -> {
});
} catch (IOException e) {
    e.printStackTrace();
} catch (TimeoutException e) {
    e.printStackTrace();
}
}
}
}

```

Recv02.java

```

package com.xxxx.work.fair.recv;

import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.DeliverCallback;

import java.io.IOException;
import java.util.concurrent.TimeoutException;

/**
 * 工作队列-公平分发-消息接收者
 */
public class Recv02 {

```



```
private final static String QUEUE_NAME = "work_fair";

public static void main(String[] args) {
    // 创建连接工厂
    ConnectionFactory factory = new ConnectionFactory();
    factory.setHost("127.0.0.1");
    factory.setPort(5672);
    factory.setUsername("shop");
    factory.setPassword("shop");
    factory.setVirtualHost("/shop");

    try {
        // 通过工厂创建连接
        final Connection connection = factory.newConnection();
        // 获取通道
        final Channel channel = connection.createChannel();
        // 指定队列
        channel.queueDeclare(QUEUE_NAME, false, false, false, null);

        /*
            限制RabbitMQ只发不超过1条的消息给同一个消费者。
            当消息处理完毕后，有了反馈，才会进行第二次发送。
        */
        int prefetchCount = 1;
        channel.basicQos(prefetchCount);

        System.out.println(" [*] waiting for messages. To exit press\nCTRL+C");
        // 获取消息
        DeliverCallback deliverCallback = (consumerTag, delivery) -> {
            String message = new String(delivery.getBody(), "UTF-8");
            System.out.println(" [x] Received02 '" + message + "'");
            // 模拟程序执行所耗时间
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            // 手动回执消息
            channel.basicAck(delivery.getEnvelope().getDeliveryTag(),
                false);
        };
        // 监听队列
        /*
            autoAck = true代表自动确认消息
            autoAck = false代表手动确认消息
        */
        boolean autoAck = false;
        channel.basicConsume(QUEUE_NAME, autoAck, deliverCallback,
            consumerTag -> {
            });
    } catch (IOException e) {
        e.printStackTrace();
    } catch (TimeoutException e) {
        e.printStackTrace();
    }
}
```

测试

运行Send

```
Run: Recv01 x Recv02 x Send (1) x
[x] Sent 'Hello World! ----- 1'
[x] Sent 'Hello World! ----- 2'
[x] Sent 'Hello World! ----- 3'
[x] Sent 'Hello World! ----- 4'
[x] Sent 'Hello World! ----- 5'
[x] Sent 'Hello World! ----- 6'
[x] Sent 'Hello World! ----- 7'
[x] Sent 'Hello World! ----- 8'
[x] Sent 'Hello World! ----- 9'
[x] Sent 'Hello World! ----- 10'
[x] Sent 'Hello World! ----- 11'
[x] Sent 'Hello World! ----- 12'
[x] Sent 'Hello World! ----- 13'
[x] Sent 'Hello World! ----- 14'
[x] Sent 'Hello World! ----- 15'
[x] Sent 'Hello World! ----- 16'
[x] Sent 'Hello World! ----- 17'
[x] Sent 'Hello World! ----- 18'
[x] Sent 'Hello World! ----- 19'
[x] Sent 'Hello World! ----- 20'
```

运行Recv

```
Run: Recv01 x Recv02 x Send x
[x] Received01 'Hello World! ----- 1'
[x] Received01 'Hello World! ----- 3'
[x] Received01 'Hello World! ----- 5'
[x] Received01 'Hello World! ----- 6'
[x] Received01 'Hello World! ----- 8'
[x] Received01 'Hello World! ----- 9'
[x] Received01 'Hello World! ----- 11'
[x] Received01 'Hello World! ----- 12'
[x] Received01 'Hello World! ----- 14'
[x] Received01 'Hello World! ----- 15'
[x] Received01 'Hello World! ----- 17'
[x] Received01 'Hello World! ----- 18'
[x] Received01 'Hello World! ----- 20'
```

```
Run: Recv01 x Recv02 x Send x
[x] Received02 'Hello World! ----- 2'
[x] Received02 'Hello World! ----- 4'
[x] Received02 'Hello World! ----- 7'
[x] Received02 'Hello World! ----- 10'
[x] Received02 'Hello World! ----- 13'
[x] Received02 'Hello World! ----- 16'
[x] Received02 'Hello World! ----- 19'
```

总结

从结果可以看出1号消费者消费消息数量明显高于2号，即消息通过fair 机制被公平分发到每个消费者。

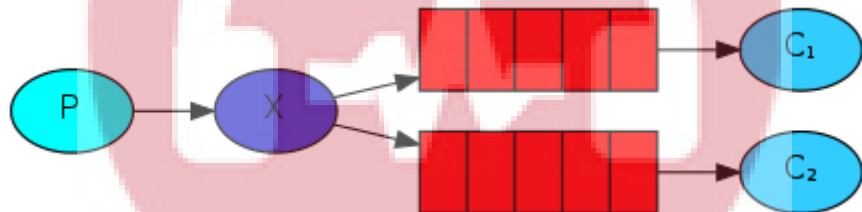
问题：生产者产生的消息只可以被一个消费者消费，可不可以被多个消费者消费呢？

解决：采用发布与订阅模式。

总结：工作队列-公平轮询分发-根据不同消费者机器硬件配置，消息处理速度不同，收到的消息数量也不同，通常速度快的处理的消息数量比较多，最大化使用计算机资源。适用于生成环境。

Publish/Subscribe-消息的发布与订阅模式队列

对于微信公众号，相信每个人都订阅过，当公众号发送新的消息后，对于订阅过该公众号的所有用户均可以收到消息，这个场景大家都能明白，同样对于RabbitMQ消息的处理也支持这种消息处理，当生产者把消息投送出去后，不同的消费者均可以对该消息进行消费，而不是消息被一个消费者消费后就立即从队列中删除，对于这种消息处理，我们通常称之为消息的发布与订阅模式，凡是消费者订阅了该消息，均能够收到对应消息进行处理，比较常见的如用户注册操作。模型图如下：



从图中看到:

消息产生后不是直接投送到队列中，而是将消息先投送给Exchange交换机，然后消息经过Exchange 交换机投递到相关队列

多个消费者消费的不再是同一个队列，而是每个消费者消费属于自己的队列。

具体实现核心代码如下：

Sending

Send.java

```
package com.xxxx.publish.subscribe.fanout.send;

import com.rabbitmq.client.BuiltinExchangeType;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;
```

```
import java.io.IOException;
import java.util.concurrent.TimeoutException;
```



```
/**
 * 发布与订阅模式队列-fanout广播模式-消息发送者
 */
public class Send {

    // 队列名称
    // 如果不声明队列，会使用默认值，RabbitMQ会创建一个排他队列，连接断开后自动删除
    //private final static String QUEUE_NAME = "ps_fanout";
    // 交换机名称
    private static final String EXCHANGE_NAME = "exchange_fanout";

    public static void main(String[] args) {
        // 创建连接工厂
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("127.0.0.1");
        factory.setPort(5672);
        factory.setUsername("shop");
        factory.setPassword("shop");
        factory.setVirtualHost("/shop");

        Connection connection = null;
        Channel channel = null;
        try {
            // 通过工厂创建连接
            connection = factory.newConnection();
            // 获取通道
            channel = connection.createChannel();
            // 声明队列
            //channel.queueDeclare(QUEUE_NAME, false, false, false, null);
            // 绑定交换机 fanout: 广播模式
            channel.exchangeDeclare(EXCHANGE_NAME, BuiltinExchangeType.FANOUT);
            // 创建消息，模拟发送手机号码和邮件地址
            String message = "18600002222|12345@qq.com";
            // 将产生的消息发送至交换机
            channel.basicPublish(EXCHANGE_NAME, "", null, message.getBytes("UTF-8"));

            System.out.println(" [x] Sent '" + message + "'");
        } catch (IOException e) {
            e.printStackTrace();
        } catch (TimeoutException e) {
            e.printStackTrace();
        } finally {
            try {
                // 关闭通道
                if (null != channel && channel.isOpen())
                    channel.close();
                // 关闭连接
                if (null != connection && connection.isOpen())
                    connection.close();
            } catch (TimeoutException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```



Receiving

这里对于消费者，消费消息时，消息通过交换机Exchange被路由到指定队列,绑定队列到指定交换机来实现，一个消费者接到消息后用于邮件发送模拟，另一消费者收到消息，用于短信发送模拟。

Recv01.java

```
package com.xxxx.publish.subscribe.fanout.recv;

import com.rabbitmq.client.*;

import java.io.IOException;
import java.util.concurrent.TimeoutException;

/**
 * 发布与订阅模式队列-fanout广播模式-消息接收者
 */
public class Recv01 {

    // 交换机名称
    private static final String EXCHANGE_NAME = "exchange_fanout";

    public static void main(String[] args) {
        // 创建连接工厂
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("127.0.0.1");
        factory.setPort(5672);
        factory.setUsername("shop");
        factory.setPassword("shop");
        factory.setVirtualHost("/shop");

        try {
            // 通过工厂创建连接
            final Connection connection = factory.newConnection();
            // 获取通道
            final Channel channel = connection.createChannel();
            // 绑定交换机 fanout: 广播模式
            channel.exchangeDeclare(EXCHANGE_NAME, BuiltInExchangeType.FANOUT);
            // 获取队列名称
            String queueName = channel.queueDeclare().getQueue();
            // 绑定队列
            channel.queueBind(queueName, EXCHANGE_NAME, "");
            /*
             限制RabbitMQ只发不超过1条的消息给同一个消费者。
             当消息处理完毕后，有了反馈，才会进行第二次发送。
            */
            int prefetchCount = 1;
            channel.basicQos(prefetchCount);

            System.out.println("[*] waiting for messages. To exit press CTRL+C");

            // 获取消息，按|分割以后一个消费者发短信，一个消费者发邮件
            DeliverCallback deliverCallback = (consumerTag, delivery) -> {
                String message = new String(delivery.getBody(), "UTF-8");
                System.out.println("[x] Received01 '" + message + "'");
                // 手动回执消息
            }
        } catch (IOException | TimeoutException e) {
            e.printStackTrace();
        }
    }
}
```



```
channel.basicAck(delivery.getEnvelope().getDeliveryTag(),
```

```
    });  
    // 监听队列  
    /*  
        autoAck = true代表自动确认消息  
        autoAck = false代表手动确认消息  
    */  
    boolean autoAck = false;  
    channel.basicConsume(queueName, autoAck, deliverCallback,  
consumerTag -> {  
        });  
    } catch (IOException e) {  
        e.printStackTrace();  
    } catch (TimeoutException e) {  
        e.printStackTrace();  
    }  
    }  
}
```

Recv02.java

```
package com.xxxx.publish.subscribe.fanout.recv;  
  
import com.rabbitmq.client.*;  
  
import java.io.IOException;  
import java.util.concurrent.TimeoutException;  
  
/**  
 * 发布与订阅模式队列-fanout广播模式-消息接收者  
 */  
public class Recv02 {  
  
    // 交换机名称  
    private static final String EXCHANGE_NAME = "exchange_fanout";  
  
    public static void main(String[] args) {  
        // 创建连接工厂  
        ConnectionFactory factory = new ConnectionFactory();  
        factory.setHost("127.0.0.1");  
        factory.setPort(5672);  
        factory.setUsername("shop");  
        factory.setPassword("shop");  
        factory.setVirtualHost("/shop");  
  
        try {  
            // 通过工厂创建连接  
            final Connection connection = factory.newConnection();  
            // 获取通道  
            final Channel channel = connection.createChannel();  
            // 绑定交换机 fanout: 广播模式  
            channel.exchangeDeclare(EXCHANGE_NAME, BuiltinExchangeType.FANOUT);  
            // 获取队列名称  
            String queueName = channel.queueDeclare().getQueue();  
            // 绑定队列  
            channel.queueBind(queueName, EXCHANGE_NAME, "");  

```



```

/*
    限制RabbitMQ只发不超过1条的消息给同一个消费者。
    当消息处理完毕后，有了反馈，才会进行第二次发送。
*/
int prefetchCount = 1;
channel.basicQos(prefetchCount);

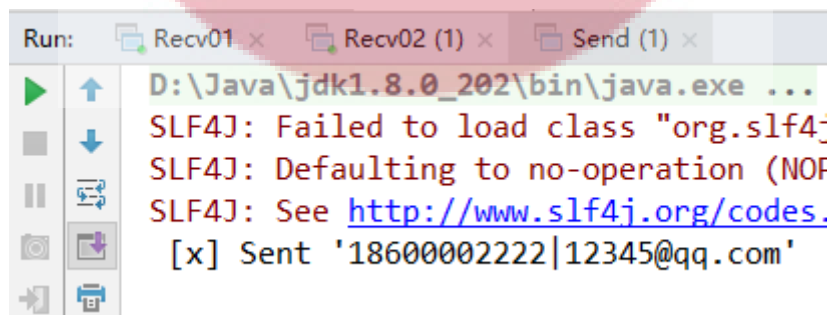
System.out.println(" [*] waiting for messages. To exit press
CTRL+C");

// 获取消息，按|分割以后一个消费者发短信，一个消费者发邮件
DeliverCallback deliverCallback = (consumerTag, delivery) -> {
    String message = new String(delivery.getBody(), "UTF-8");
    System.out.println(" [x] Received01 '" + message + "'");
    // 手动回执消息
    channel.basicAck(delivery.getEnvelope().getDeliveryTag(),
false);
};
// 监听队列
/*
    autoAck = true代表自动确认消息
    autoAck = false代表手动确认消息
*/
boolean autoAck = false;
channel.basicConsume(queueName, autoAck, deliverCallback,
consumerTag -> {
});
} catch (IOException e) {
    e.printStackTrace();
} catch (TimeoutException e) {
    e.printStackTrace();
}
}
}
}

```

测试

运行Send

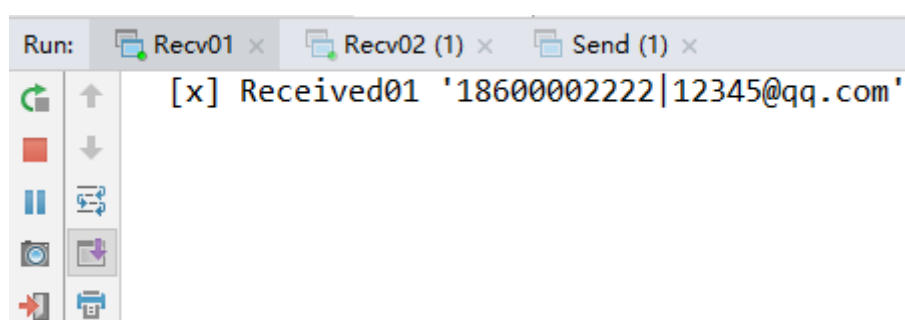


```

Run: Recv01 x Recv02 (1) x Send (1) x
D:\Java\jdk1.8.0_202\bin\java.exe ...
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder"
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#noLoggerBinder for further details.
[x] Sent '18600002222|12345@qq.com'

```

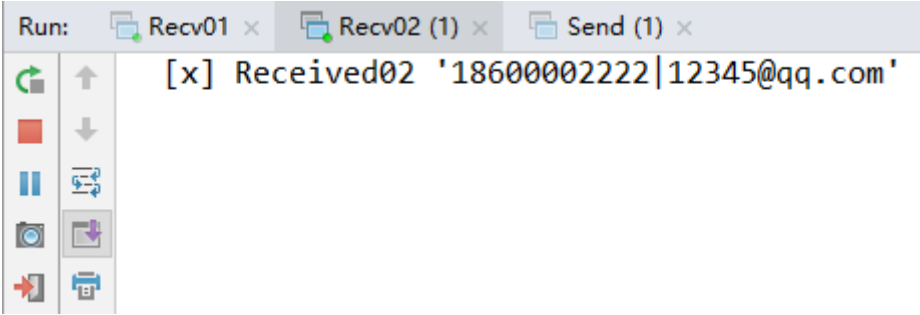
运行Recv



```

Run: Recv01 x Recv02 (1) x Send (1) x
[x] Received01 '18600002222|12345@qq.com'

```



Virtual host	Name	Type	Features	Message rate in	Message rate out	+/-
/	(AMQP default)	direct	D			
/	amq.direct	direct	D			
/	amq.fanout	fanout	D			
/	amq.headers	headers	D			
/	amq.match	headers	D			
/	amq.rabbitmq.trace	topic	D I			
/	amq.topic	topic	D			
/	smsExchange	topic	D			
/shop	(AMQP default)	direct	D	0.00/s	0.00/s	
/shop	amq.direct	direct	D			
/shop	amq.fanout	fanout	D			
/shop	amq.headers	headers	D			
/shop	amq.match	headers	D			
/shop	amq.rabbitmq.trace	topic	D I			
/shop	amq.topic	topic	D			
/shop	exchange_fanout	fanout		0.00/s	0.00/s	

总结

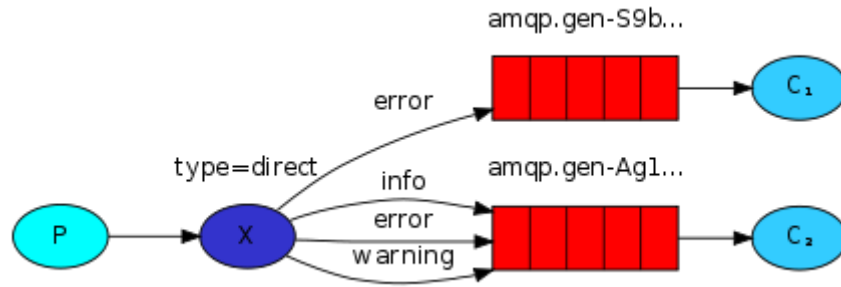
从结果可以看出生产者发送了一条消息，用于邮件发送和短信发送的消费者均可以收到消息进行后续处理。

问题：生产者产生的消息所有消费者都可以消费，可不可以指定某些消费者消费呢？

解决：采用direct路由模式。

Routing-路由模式队列

通过案例03,可以看到，生产者将消息投送给交换机后，消息经交换机分发到不同的队列即:交换机收到消息，默认对于绑定到每个交换机的队列均会接收到交换机分发的消息，对于案例03的交换机的消息分发Exchange Types为 fanout 类型，通常在真正项目开发时会遇到这种情况:在对项目信息输出日志进行收集时，会把日志(error warning,info)分类进行输出，这时通过Exchange Types中的 direct 类型就可以实现，针对不同的消息，在对消息进行消费时，通过 Exchange types 以及 Routing key 设置的规则，便可以将不同消息路由到不同的队列中然后交给不同消费者进行消费操作。模型图如下：



从图中可以看出:

1. 生产者产生的消息投给交换机
2. 交换机投送消息时的Exchange Types为direct类型
3. 消息通过定义的Routing Key被路由到指定的队列进行后续消费

具体实现核心代码如下:

Sending

Send.java

```
package com.xxxx.publish.subscribe.direct.send;

import com.rabbitmq.client.BuiltinExchangeType;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;

import java.io.IOException;
import java.util.concurrent.TimeoutException;

/**
 * direct路由模式队列-消息发送者
 */
public class Send {

    // 队列名称
    // 如果不声明队列, 会使用默认值, RabbitMQ会创建一个排他队列, 连接断开后自动删除
    // 交换机名称
    private static final String EXCHANGE_NAME = "exchange_direct";

    public static void main(String[] args) {
        // 创建连接工厂
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("127.0.0.1");
        factory.setPort(5672);
        factory.setUsername("shop");
        factory.setPassword("shop");
        factory.setVirtualHost("/shop");

        Connection connection = null;
        Channel channel = null;
        try {
            // 通过工厂创建连接
            connection = factory.newConnection();
            // 获取通道
```

```
channel = connection.createChannel();
// 绑定交换机 direct: 路由模式
channel.exchangeDeclare(EXCHANGE_NAME, BuiltinExchangeType.DIRECT);
// 创建消息, 模拟收集不同级别日志
String message = "INFO消息";
//String message = "WARNING消息";
//String message = "ERROR消息";
// 设置路由routingKey
String routingKey = "info";
//String routingKey = "error";
// 将产生的消息发送至交换机
channel.basicPublish(EXCHANGE_NAME, routingKey, null,
message.getBytes("UTF-8"));
System.out.println(" [x] Sent '" + message + "'");
} catch (IOException e) {
    e.printStackTrace();
} catch (TimeoutException e) {
    e.printStackTrace();
} finally {
    try {
        // 关闭通道
        if (null != channel && channel.isOpen())
            channel.close();
        // 关闭连接
        if (null != connection && connection.isOpen())
            connection.close();
    } catch (TimeoutException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}
```

Receiving

消费者对消息进行后续消费时, 对于接收消息的队列在对消息进行接收时, 绑定到每一个交换机上的队列均会指定其Routing Key规则, 通过路由规则将消息路由到执行队列中。

消费者01 routingKey=info和warning, 对应级别日志消息均会路由到该队列中。

Recv01.java

```
package com.xxxx.publish.subscribe.direct.recv;

import com.rabbitmq.client.*;

import java.io.IOException;
import java.util.concurrent.TimeoutException;

/**
 * direct路由模式队列-消息接收者
 */
public class Recv01 {

    // 交换机名称
```



```
private static final String EXCHANGE_NAME = "exchange_direct";

public static void main(String[] args) {
    // 创建连接工厂
    ConnectionFactory factory = new ConnectionFactory();
    factory.setHost("127.0.0.1");
    factory.setPort(5672);
    factory.setUsername("shop");
    factory.setPassword("shop");
    factory.setVirtualHost("/shop");

    try {
        // 通过工厂创建连接
        final Connection connection = factory.newConnection();
        // 获取通道
        final Channel channel = connection.createChannel();
        // 绑定交换机 direct: 路由模式
        channel.exchangeDeclare(EXCHANGE_NAME, BuiltinExchangeType.DIRECT);
        // 获取队列名称
        String queueName = channel.queueDeclare().getQueue();
        // 设置路由routingKey
        String routingKeyInfo = "info";
        String routingKeyWarning = "warning";
        // 绑定队列
        channel.queueBind(queueName, EXCHANGE_NAME, routingKeyInfo);
        channel.queueBind(queueName, EXCHANGE_NAME, routingKeyWarning);
        /*
            限制RabbitMQ只发不超过1条的消息给同一个消费者。
            当消息处理完毕后，有了反馈，才会进行第二次发送。
        */
        int prefetchCount = 1;
        channel.basicQos(prefetchCount);

        System.out.println("[*] waiting for messages. To exit press
CTRL+C");

        // 获取消息，按|分割以后一个消费者发短信，一个消费者发邮件
        DeliverCallback deliverCallback = (consumerTag, delivery) -> {
            String message = new String(delivery.getBody(), "UTF-8");
            System.out.println("[x] Received01 '" + message + "'");
            // 手动回执消息
            channel.basicAck(delivery.getEnvelope().getDeliveryTag(),
false);
        };
        // 监听队列
        /*
            autoAck = true代表自动确认消息
            autoAck = false代表手动确认消息
        */
        boolean autoAck = false;
        channel.basicConsume(queueName, autoAck, deliverCallback,
consumerTag -> {
        });
    } catch (IOException e) {
        e.printStackTrace();
    } catch (TimeoutException e) {
        e.printStackTrace();
    }
}
```

消费者02 routingKey=error, 对应级别日志消息均会路由到该队列中。

Recv02.java

```
package com.xxxx.publish.subscribe.direct.recv;

import com.rabbitmq.client.*;

import java.io.IOException;
import java.util.concurrent.TimeoutException;

/**
 * direct路由模式队列-消息接收者
 */
public class Recv02 {

    // 交换机名称
    private static final String EXCHANGE_NAME = "exchange_direct";

    public static void main(String[] args) {
        // 创建连接工厂
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("127.0.0.1");
        factory.setPort(5672);
        factory.setUsername("shop");
        factory.setPassword("shop");
        factory.setVirtualHost("/shop");

        try {
            // 通过工厂创建连接
            final Connection connection = factory.newConnection();
            // 获取通道
            final Channel channel = connection.createChannel();
            // 绑定交换机 direct: 路由模式
            channel.exchangeDeclare(EXCHANGE_NAME, BuiltinExchangeType.DIRECT);
            // 获取队列名称
            String queueName = channel.queueDeclare().getQueue();
            // 设置路由routingKey
            String routingKey = "error";
            // 绑定队列
            channel.queueBind(queueName, EXCHANGE_NAME, routingKey);
            /*
             限制RabbitMQ只发不超过1条的消息给同一个消费者。
             当消息处理完毕后, 有了反馈, 才会进行第二次发送。
            */
            int prefetchCount = 1;
            channel.basicQos(prefetchCount);

            System.out.println(" [*] waiting for messages. To exit press CTRL+C");

            // 获取消息
            DeliverCallback deliverCallback = (consumerTag, delivery) -> {
                String message = new String(delivery.getBody(), "UTF-8");
                System.out.println(" [x] Received02 '" + message + "'");
                // 手动回执消息
            }
        } catch (IOException | TimeoutException | InterruptedException | java.util.concurrent.ExecutionException e) {
            e.printStackTrace();
        }
    }
}
```

```
channel.basicAck(delivery.getEnvelope().getDeliveryTag(),
```

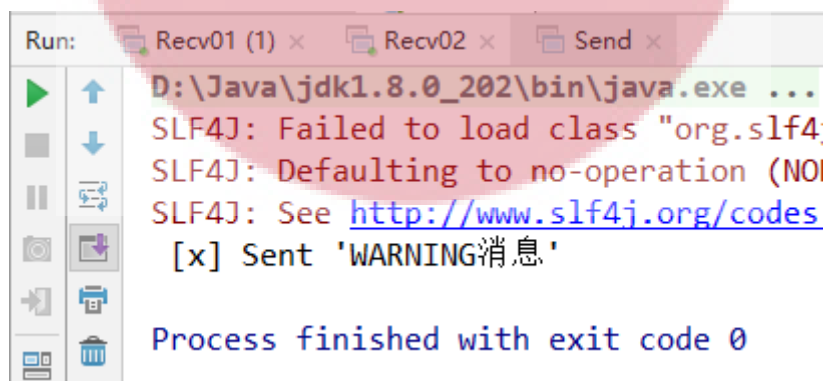
```
});
// 监听队列
/*
    autoAck = true代表自动确认消息
    autoAck = false代表手动确认消息
*/
boolean autoAck = false;
channel.basicConsume(queueName, autoAck, deliverCallback,
consumerTag -> {
    });
} catch (IOException e) {
    e.printStackTrace();
} catch (TimeoutException e) {
    e.printStackTrace();
}
}
}
```

测试

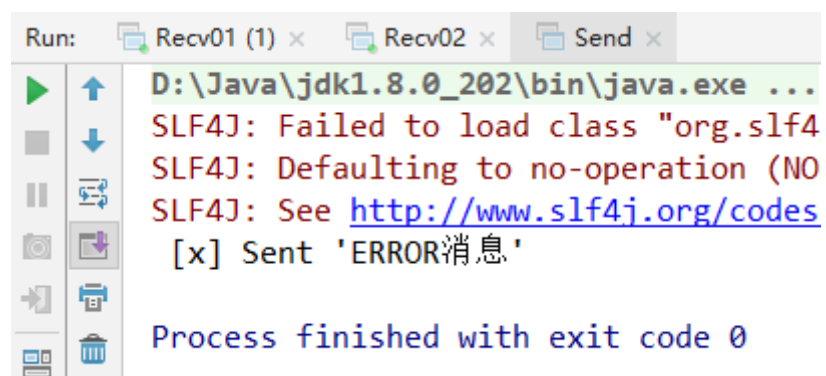
运行Send



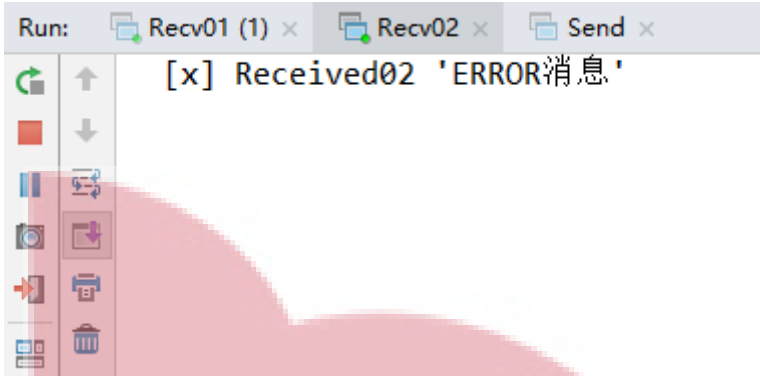
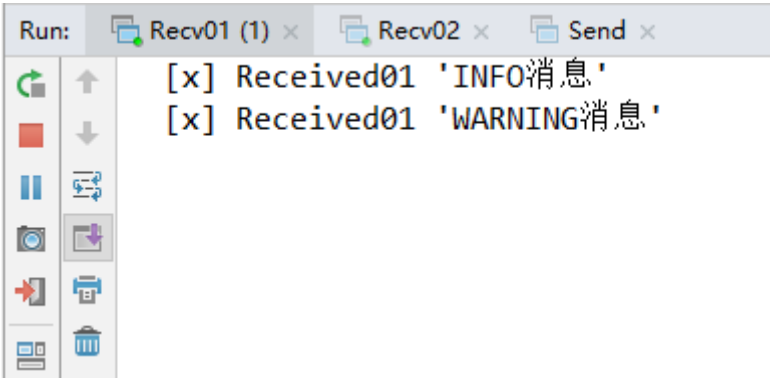
```
Run: Recv01 (1) x Recv02 x Send x
D:\Java\jdk1.8.0_202\bin\java.exe ...
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder"
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#no\_logger for further details.
[x] Sent 'INFO消息'
Process finished with exit code 0
```



```
Run: Recv01 (1) x Recv02 x Send x
D:\Java\jdk1.8.0_202\bin\java.exe ...
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder"
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#no\_logger for further details.
[x] Sent 'WARNING消息'
Process finished with exit code 0
```



```
Run: Recv01 (1) x Recv02 x Send x
D:\Java\jdk1.8.0_202\bin\java.exe ...
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder"
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#no\_logger for further details.
[x] Sent 'ERROR消息'
Process finished with exit code 0
```



Virtual host	Name	Type	Features	Message rate in	Message rate out	+/-
/	(AMQP default)	direct	D			
/	amq.direct	direct	D			
/	amq.fanout	fanout	D			
/	amq.headers	headers	D			
/	amq.match	headers	D			
/	amq.rabbitmq.trace	topic	D I			
/	amq.topic	topic	D			
/	smsExchange	topic	D			
/shop	(AMQP default)	direct	D	0.00/s	0.00/s	
/shop	amq.direct	direct	D			
/shop	amq.fanout	fanout	D			
/shop	amq.headers	headers	D			
/shop	amq.match	headers	D			
/shop	amq.rabbitmq.trace	topic	D I			
/shop	amq.topic	topic	D			
/shop	exchange_direct	direct		0.00/s	0.00/s	
/shop	exchange_fanout	fanout		0.00/s	0.00/s	

总结

从结果可以看出生产者发送了多条设置了路由规则的消息，消费者可以根据具体的路由规则消费对应队列中的消息，而不是所有消费者都可以消费所有消息了。

问题：生产者产生的消息如果场景需求过多需要设置很多路由规则，可不可以减少？

解决：采用topic主题模式。

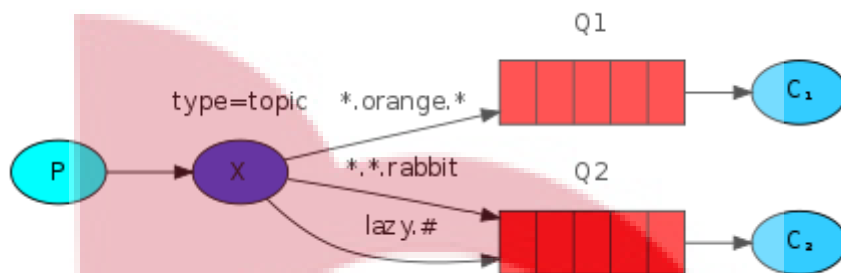
Topics-主题模式队列



通过案例04看到消息通过交换机Exchange Type以及Routing Key规则，可以将消息路由到指定的队列，也符合在工作中的场景去使用的一种方式，对于RabbitMq 除了 direct 模式外，Mq 同样还提供了 topics 主题模式来对消息进行匹配路由，比如在项目开发中，拿商品模块来说，对于商品的查询功能在对商品进行查询时我们将查询消息路由到查询对应队列，而对于商品的添加、更新、删除等操作我们统一路由到另外一个队列来进行处理，此时采用direct 模式可以实现，但对于维护的队列可能就不太容易进行维护，如果涉及模块很多，此时对应队列数量就很多，此时我们就可以通过 topic 主题模式来对消息路由时进行匹配，通过指定的匹配模式将消息路由到匹配到的队列中进行后续处理。对于 routing key匹配模式定义规则举例如下：

routing key为一个句点号 . 分隔的字符串（我们将被句点号 . 分隔开的每一段独立的字符串称为一个单词），如“stock.usd.nyse”、“nyse.vmw”、“quick.orange.rabbit”
routing key中可以存在两种特殊字符 * 与 #，用于做模糊匹配，其中 * 用于匹配一个单词，# 用于匹配多个单词（可以是零个）

例如：



以上图中的配置为例：

routingKey="quick.orange.rabbit"的消息会同时路由到Q1与Q2，
routingKey="lazy.orange.fox"的消息会路由到Q1，Q2，
routingKey="lazy.brown.fox"的消息会路由到Q2，
routingKey="lazy.pink.rabbit"的消息会路由到Q2；
routingKey="quick.brown.fox"；
routingKey="orange"；
routingKey="quick.orange.male.rabbit"的消息将会被丢弃，因为它们没有匹配任何 bindingKey。

具体实现核心代码：

Sending

这里以商品模块为例，商品查询路由到商品查询队列，商品更新路由到商品更新(添加，更新，删除操作)队列中。

Send.java

```
package com.xxxx.publish.subscribe.topic.send;

import com.rabbitmq.client.BuiltinExchangeType;
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 Send {

    // 队列名称
    // 如果不声明队列，会使用默认值，RabbitMQ会创建一个排他队列，连接断开后自动删除
    // 交换机名称
    private static final String EXCHANGE_NAME = "exchange_topic";

    public static void main(String[] args) {
        // 创建连接工厂
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("127.0.0.1");
        factory.setPort(5672);
        factory.setUsername("shop");
        factory.setPassword("shop");
        factory.setVirtualHost("/shop");

        Connection connection = null;
        Channel channel = null;
        try {
            // 通过工厂创建连接
            connection = factory.newConnection();
            // 获取通道
            channel = connection.createChannel();
            // 绑定交换机 topic: 主题模式
            channel.exchangeDeclare(EXCHANGE_NAME, BuiltinExchangeType.TOPIC);
            // 创建消息，模拟商品模块
            String message = "商品查询操作";
            //String message = "商品更新操作";
            // 设置路由routingKey
            String routingKey = "select.goods.byId";
            //String routingKey = "update.goods.byId.andName";
            // 将产生的消息发送至交换机
            channel.basicPublish(EXCHANGE_NAME, routingKey, null,
message.getBytes("UTF-8"));
            System.out.println(" [x] Sent '" + message + "'");
        } catch (IOException e) {
            e.printStackTrace();
        } catch (TimeoutException e) {
            e.printStackTrace();
        } finally {
            try {
                // 关闭通道
                if (null != channel && channel.isOpen())
                    channel.close();
                // 关闭连接
                if (null != connection && connection.isOpen())
                    connection.close();
            } catch (TimeoutException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```



Recv01.java

```
package com.xxxx.publish.subscribe.topic.recv;

import com.rabbitmq.client.*;

import java.io.IOException;
import java.util.concurrent.TimeoutException;

/**
 * topic主题模式队列-消息接收者
 */
public class Recv01 {

    // 交换机名称
    private static final String EXCHANGE_NAME = "exchange_topic";

    public static void main(String[] args) {
        // 创建连接工厂
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("127.0.0.1");
        factory.setPort(5672);
        factory.setUsername("shop");
        factory.setPassword("shop");
        factory.setVirtualHost("/shop");

        try {
            // 通过工厂创建连接
            final Connection connection = factory.newConnection();
            // 获取通道
            final Channel channel = connection.createChannel();
            // 绑定交换机 topic: 主题模式
            channel.exchangeDeclare(EXCHANGE_NAME, BuiltinExchangeType.TOPIC);
            // 获取队列名称
            String queueName = channel.queueDeclare().getQueue();
            // 设置路由routingKey
            String routingKey = "select.goods.*";
            // 绑定队列
            channel.queueBind(queueName, EXCHANGE_NAME, routingKey);
            /*
             限制RabbitMQ只发不超过1条的消息给同一个消费者。
             当消息处理完毕后，有了反馈，才会进行第二次发送。
            */
            int prefetchCount = 1;
            channel.basicQos(prefetchCount);

            System.out.println(" [*] waiting for messages. To exit press CTRL+C");

            // 获取消息，按|分割以后一个消费者发短信，一个消费者发邮件
            DeliverCallback deliverCallback = (consumerTag, delivery) -> {
                String message = new String(delivery.getBody(), "UTF-8");
                System.out.println(" [x] Received01 '" + message + "'");
                // 手动回执消息
                channel.basicAck(delivery.getEnvelope().getDeliveryTag(),
                    false);
            };
        } catch (IOException | TimeoutException e) {
            e.printStackTrace();
        }
    }
}
```



```
};  
// 监听队列  
/*  
    autoAck = true代表自动确认消息  
    autoAck = false代表手动确认消息  
*/  
boolean autoAck = false;  
channel.basicConsume(queueName, autoAck, deliverCallback,  
consumerTag -> {  
    });  
    } catch (IOException e) {  
        e.printStackTrace();  
    } catch (TimeoutException e) {  
        e.printStackTrace();  
    }  
}  
}
```

Recv02.java

```
package com.xxxx.publish.subscribe.topic.recv;  
  
import com.rabbitmq.client.*;  
  
import java.io.IOException;  
import java.util.concurrent.TimeoutException;  
  
/**  
 * topic主题模式队列-消息接收者  
 */  
public class Recv02 {  
  
    // 交换机名称  
    private static final String EXCHANGE_NAME = "exchange_topic";  
  
    public static void main(String[] args) {  
        // 创建连接工厂  
        ConnectionFactory factory = new ConnectionFactory();  
        factory.setHost("127.0.0.1");  
        factory.setPort(5672);  
        factory.setUsername("shop");  
        factory.setPassword("shop");  
        factory.setVirtualHost("/shop");  
  
        try {  
            // 通过工厂创建连接  
            final Connection connection = factory.newConnection();  
            // 获取通道  
            final Channel channel = connection.createChannel();  
            // 绑定交换机 topic: 主题模式  
            channel.exchangeDeclare(EXCHANGE_NAME, BuiltinExchangeType.TOPIC);  
            // 获取队列名称  
            String queueName = channel.queueDeclare().getQueue();  
            // 设置路由routingKey  
            String routingKey = "update.goods.#";  
            // 绑定队列  
            channel.queueBind(queueName, EXCHANGE_NAME, routingKey);  

```

```

/*
    限制RabbitMQ只发不超过1条的消息给同一个消费者。
    当消息处理完毕后，有了反馈，才会进行第二次发送。
*/
int prefetchCount = 1;
channel.basicQos(prefetchCount);

System.out.println(" [*] waiting for messages. To exit press
CTRL+C");

// 获取消息
DeliverCallback deliverCallback = (consumerTag, delivery) -> {
    String message = new String(delivery.getBody(), "UTF-8");
    System.out.println(" [x] Received02 '" + message + "'");
    // 手动回执消息
    channel.basicAck(delivery.getEnvelope().getDeliveryTag(),
false);
};

// 监听队列
/*
    autoAck = true代表自动确认消息
    autoAck = false代表手动确认消息
*/
boolean autoAck = false;
channel.basicConsume(queueName, autoAck, deliverCallback,
consumerTag -> {
});
} catch (IOException e) {
    e.printStackTrace();
} catch (TimeoutException e) {
    e.printStackTrace();
}
}
}
}

```

测试

运行Send



```

Run: Recv01 x Recv02 (1) x Send (1) x
D:\Java\jdk1.8.0_202\bin\java.exe ...
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder"
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#no\_logger for further details.
[x] Sent '商品查询操作'

```

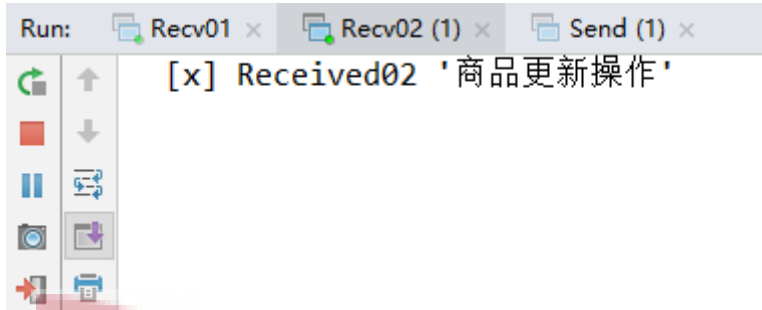
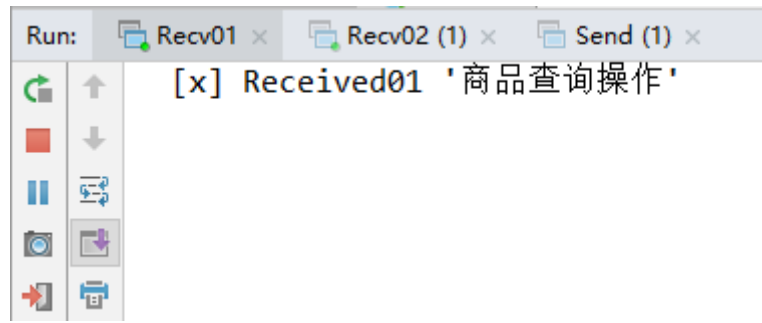


```

Run: Recv01 x Recv02 (1) x Send (1) x
D:\Java\jdk1.8.0_202\bin\java.exe ...
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder"
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#no\_logger for further details.
[x] Sent '商品更新操作'

```

运行Recv



Virtual host	Name	Type	Features	Message rate in	Message rate out	+/-
/	(AMQP default)	direct	D			
/	amq.direct	direct	D			
/	amq.fanout	fanout	D			
/	amq.headers	headers	D			
/	amq.match	headers	D			
/	amq.rabbitmq.trace	topic	D I			
/	amq.topic	topic	D			
/	smsExchange	topic	D			
/shop	(AMQP default)	direct	D	0.00/s	0.00/s	
/shop	amq.direct	direct	D			
/shop	amq.fanout	fanout	D			
/shop	amq.headers	headers	D			
/shop	amq.match	headers	D			
/shop	amq.rabbitmq.trace	topic	D I			
/shop	amq.topic	topic	D			
/shop	exchange_direct	direct		0.00/s	0.00/s	
/shop	exchange_fanout	fanout		0.00/s	0.00/s	
/shop	exchange_topic	topic		0.00/s	0.00/s	

总结

从结果可以看出生产者发送了多条设置了路由匹配规则(主题)的消息，根据不同的路由匹配规则(主题)，可以将消息根据指定的routing key路由到匹配到的队列中，也是在生产中比较常见的一种消息处理方式。

问题：RabbitMQ本身是基于异步的消息处理，是否可以同步实现？

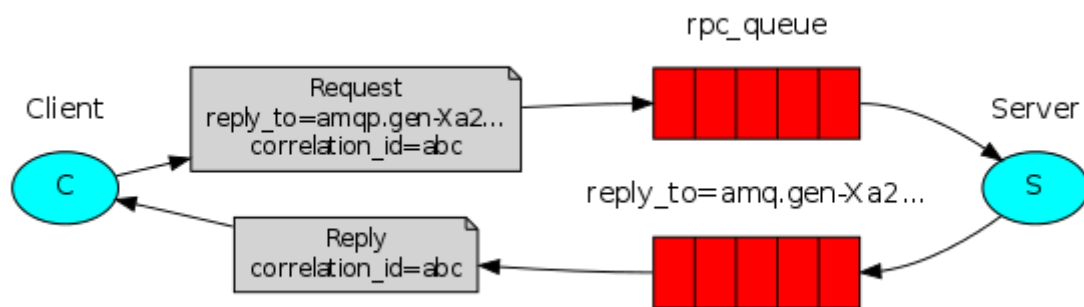
解决：采用RPC模式。

RPC-远程过程调用模式队列

MQ本身是基于异步的消息处理，前面的示例中所有的生产者（P）将消息发送到RabbitMQ后不会知道消费者（C）处理成功或者失败（甚至连有没有消费者来处理这条消息都不知道）。



但在实际的应用场景中，我们很可能需要一些同步处理，需要同步等待服务端将我的消息处理完成后再进行下一步处理。这相当于RPC（Remote Procedure Call，远程过程调用）。在RabbitMQ中也支持RPC。



RabbitMQ中实现RPC的机制是：

1. 客户端发送请求（消息）时，在消息的属性（MessageProperties，在AMQP协议中定义了14种properties，这些属性会随着消息一起发送）中设置两个值 replyTo（一个Queue名称，用于告诉服务器处理完成后将通知我的消息发送到这个Queue中）和 correlationId（此次请求的标识号，服务器处理完成后需要将此属性返还，客户端将根据这个id了解哪条请求被成功执行了或执行失败）
2. 服务器端收到消息并处理
3. 服务器端处理完消息后，将生成一条应答消息到 replyTo 指定的Queue，同时携带 correlationId 属性

客户端之前已订阅 replyTo 指定的Queue，从中收到服务器的应答消息后，根据其中的 correlationId 属性分析哪条请求被执行了，根据执行结果进行后续业务处理。

具体实现核心代码：

Server

Server.java

```
package com.xxxx.rpc.server;

import com.rabbitmq.client.*;

import java.io.IOException;
import java.util.concurrent.TimeoutException;

/**
 * RPC模式队列-服务端
 */
public class RPCServer {

    // 队列名称
    private static final String RPC_QUEUE_NAME = "rpc_queue";

    /**
     * 计算斐波那契数列
     *
     * @param n
     * @return
     */
}
```



```
private static int fib(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fib(n - 1) + fib(n - 2);
}

public static void main(String[] args) {
    // 创建连接工厂
    ConnectionFactory factory = new ConnectionFactory();
    factory.setHost("127.0.0.1");
    factory.setPort(5672);
    factory.setUsername("shop");
    factory.setPassword("shop");
    factory.setVirtualHost("/shop");

    try {
        // 通过工厂创建连接
        final Connection connection = factory.newConnection();
        // 获取通道
        final Channel channel = connection.createChannel();
        // 声明队列
        channel.queueDeclare(RPC_QUEUE_NAME, false, false, false, null);
        channel.queuePurge(RPC_QUEUE_NAME);

        /*
            限制RabbitMQ只发不超过1条的消息给同一个消费者。
            当消息处理完毕后，有了反馈，才会进行第二次发送。
        */
        int prefetchCount = 1;
        channel.basicQos(prefetchCount);

        System.out.println(" [x] Awaiting RPC requests");

        Object monitor = new Object();
        // 获取消息
        DeliverCallback deliverCallback = (consumerTag, delivery) -> {
            // 获取replyTo队列和correlationId请求标识
            AMQP.BasicProperties replyProps = new AMQP.BasicProperties
                .Builder()
                    .correlationId(delivery.getProperties().getCorrelationId())
                    .build();

            String response = "";
            try {
                // 接收客户端消息
                String message = new String(delivery.getBody(), "UTF-8");
                int n = Integer.parseInt(message);

                System.out.println(" [.] fib(" + message + ")");
                // 服务端根据业务需求处理
                response += fib(n);
            } catch (RuntimeException e) {
                System.out.println(" [.] " + e.toString());
            } finally {
                // 将处理结果发送至replyTo队列同时携带correlationId属性
                channel.basicPublish("",
                    delivery.getProperties().getReplyTo(), replyProps,

```




```
        response.getBytes("UTF-8"));
        // 手动回执消息
        channel.basicAck(delivery.getEnvelope().getDeliveryTag(),
false);

        // RabbitMq consumer worker thread notifies the RPC server
owner thread

        // RabbitMq消费者工作线程通知RPC服务器其他所有线程运行
        synchronized (monitor) {
            monitor.notify();
        }
    }
};
// 监听队列
/*
    autoAck = true代表自动确认消息
    autoAck = false代表手动确认消息
*/
boolean autoAck = false;
channel.basicConsume(RPC_QUEUE_NAME, autoAck, deliverCallback,
consumerTag -> {
});
// wait and be prepared to consume the message from RPC client.
// 线程等待并准备接收来自RPC客户端的消息
while (true) {
    synchronized (monitor) {
        try {
            monitor.wait();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
} catch (IOException e) {
    e.printStackTrace();
} catch (TimeoutException e) {
    e.printStackTrace();
}
}
}
```

Client

Client.java

```
package com.xxxx.rpc.client;

import com.rabbitmq.client.AMQP;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;

import java.io.IOException;
import java.util.UUID;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.TimeoutException;
```



* RPC模式队列-客户端

*/

```
public class RPCCClient implements AutoCloseable {

    private Connection connection;
    private Channel channel;
    // 队列名称
    private String requestQueueName = "rpc_queue";

    // 初始化连接
    public RPCCClient() throws IOException, TimeoutException {
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("127.0.0.1");
        factory.setPort(5672);
        factory.setUsername("shop");
        factory.setPassword("shop");
        factory.setVirtualHost("/shop");

        connection = factory.newConnection();
        channel = connection.createChannel();
    }

    public static void main(String[] args) {
        try (RPCCClient fibonacciRpc = new RPCCClient()) {
            for (int i = 0; i < 10; i++) {
                String i_str = Integer.toString(i);
                System.out.println(" [x] Requesting fib(" + i_str + ")");
                // 请求服务端
                String response = fibonacciRpc.call(i_str);
                System.out.println(" [.] Got '" + response + "'");
            }
        } catch (IOException | TimeoutException | InterruptedException e) {
            e.printStackTrace();
        }
    }

    // 请求服务端
    public String call(String message) throws IOException, InterruptedException {

        // correlationId请求标识ID
        final String corrId = UUID.randomUUID().toString();

        // 获取队列名称
        String replyQueueName = channel.queueDeclare().getQueue();

        // 设置replyTo队列和correlationId请求标识
        AMQP.BasicProperties props = new AMQP.BasicProperties
            .Builder()
            .correlationId(corrId)
            .replyTo(replyQueueName)
            .build();

        // 发送消息至队列
        channel.basicPublish("", requestQueueName, props, message.getBytes("UTF-8"));

        // 设置线程等待，每次只接收一个响应结果
    }
}
```



```
final BlockingQueue<String> response = new ArrayBlockingQueue<>(1);

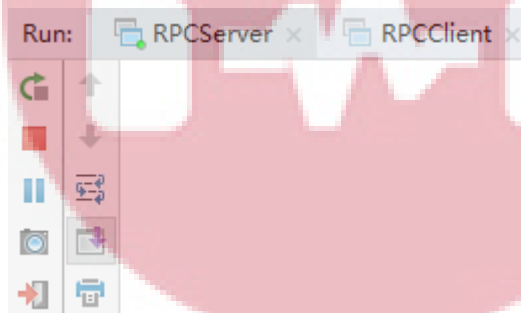
// 接受服务器返回结果
String ctag = channel.basicConsume(replyQueueName, true, (consumerTag,
delivery) -> {
    if (delivery.getProperties().getCorrelationId().equals(corrId)) {
        // 将给定的元素在给定的时间内设置到线程队列中，如果设置成功返回true，否则返
        回false
        response.offer(new String(delivery.getBody(), "UTF-8"));
    }
}, consumerTag -> {
});

// 从线程队列中获取值，如果线程队列中没有值，线程会一直阻塞，直到线程队列中有值，并且
取得该值
String result = response.take();
// 从消息队列中丢弃该值
channel.basicCancel(ctag);
return result;
}

// 关闭连接
public void close() throws IOException {
    connection.close();
}
}
```

测试

运行Server



运行Client

```
Run: RPCServer x RPCClient x
[x] Requesting fib(0)
[.] Got '0'
[x] Requesting fib(1)
[.] Got '1'
[x] Requesting fib(2)
[.] Got '1'
[x] Requesting fib(3)
[.] Got '2'
[x] Requesting fib(4)
[.] Got '3'
[x] Requesting fib(5)
[.] Got '5'
[x] Requesting fib(6)
[.] Got '8'
[x] Requesting fib(7)
[.] Got '13'
[x] Requesting fib(8)
[.] Got '21'
[x] Requesting fib(9)
[.] Got '34'
```

查看Server

```
Run: RPCServer x RPCClient x
[.] fib(0)
[.] fib(1)
[.] fib(2)
[.] fib(3)
[.] fib(4)
[.] fib(5)
[.] fib(6)
[.] fib(7)
[.] fib(8)
[.] fib(9)
```

RabbitMQ消息的事务机制

在使用RabbitMQ的时候，我们可以通过消息持久化操作来解决因为服务器的异常奔溃导致的消息丢失，除此之外我们还会遇到一个问题，当消息的发布者在将消息发送出去之后，消息到底有没有正确到达broker代理服务器呢？如果不进行特殊配置的话，默认情况下发布操作是不会返回任何信息给生产者的，也就是默认情况下我们的生产者是不知消息有没有正确到达broker的，如果在消息到达broker之前已经丢失的话，持久化操作也解决不了这个问题，因为消息根本就没到达代理服务器，你怎么进行持久化，那么这个问题该怎么解决呢？

RabbitMQ为我们提供了两种方式：

- 通过AMQP事务机制实现，这也是AMQP协议层面提供的解决方案；
- 通过将channel设置成confirm模式来实现；



RabbitMQ中与事务机制有关的方法有三个：txSelect(), txCommit() 以及 txRollback(), txSelect() 用于将当前channel设置成transaction模式, txCommit() 用于提交事务, txRollback() 用于回滚事务, 在通过 txSelect() 开启事务之后, 我们便可以发布消息给broker代理服务器了, 如果 txCommit() 提交成功了, 则消息一定到达了broker了, 如果在 txCommit() 执行之前 broker异常崩溃或者由于其他原因抛出异常, 这个时候我们便可以捕获异常通过 txRollback() 回滚事务。

SendTx.java

```
try {
    // 通过工厂创建连接
    connection = factory.newConnection();
    // 获取通道
    channel = connection.createChannel();
    // 开启事务
    channel.txSelect();
    // 声明队列
    channel.queueDeclare(QUEUE_NAME, false, false, false, null);
    // 创建消息
    String message = "Hello world!";
    // 将产生的消息放入队列
    channel.basicPublish("", QUEUE_NAME, null, message.getBytes("UTF-8"));
    System.out.println(" [x] Sent '" + message + "'");
    // 模拟程序异常
    int i = 1 / 0;
    // 提交事务
    channel.txCommit();
} catch (IOException | TimeoutException e) {
    e.printStackTrace();
    try {
        // 回滚事务
        channel.txRollback();
    } catch (IOException e1) {
        e1.printStackTrace();
    }
}
```

事务确实能够解决producer与broker之间消息确认的问题, 只有消息成功被broker接受, 事务提交才能成功, 否则我们便可以在捕获异常进行事务回滚操作同时进行消息重发, 但是使用事务机制的话会降低RabbitMQ的性能, 那么有没有更好的方法既能保障producer知道消息已经正确送到, 又能基本上不带来性能上的损失呢? 从AMQP协议的层面看是没有更好的方法, 但是RabbitMQ提供了一个更好的方案, 即将channel信道设置成confirm模式。

confirm确认模式

通过AMQP协议层面为我们提供了事务机制解决了这个问题, 但是采用事务机制实现会降低RabbitMQ的消息吞吐量, 此时处理AMQP协议层面能够实现消息事物控制外, 我们还有第二种方式即:Confirm模式。

Confirm确认模式原理



生产者将信道设置成confirm模式，一旦信道进入confirm模式，所有在该信道上发布的信息都会被指派一个唯一的ID(从1开始)，一旦消息被投递到所有匹配的队列之后，broker就会发送一个确认给生产者(包含消息的唯一ID)，这就使得生产者知道消息已经正确到达目的队列了，如果消息和队列是可持久化的，那么确认消息会将消息写入磁盘之后发出，broker回传给生产者的确认消息中deliver-tag域包含了确认消息的序列号，此外broker也可以设置basic.ack的multiple域，表示到这个序列号之前的所有消息都已经得到了处理。

confirm模式最大的好处在于他是异步的，一旦发布一条消息，生产者应用程序就可以在等信道返回确认的同时继续发送下一条消息，当消息最终得到确认之后，生产者应用便可以通过回调方法来处理该确认消息，如果RabbitMQ因为自身内部错误导致消息丢失，就会发送一条nack消息，生产者应用程序同样可以在回调方法中处理该nack消息。

在channel 被设置成 confirm 模式之后，所有被 publish 的后续消息都将被 confirm (即 ack) 或者被nack一次。但是没有对消息被 confirm 的快慢做任何保证，并且同一条消息不会既被 confirm又被nack。

注意:两种事物控制形式不能同时开启!

Confirm确认机制代码实现

实现生产者confirm 机制有三种方式:

普通confirm模式: 每发送一条消息后, 调用waitForConfirms()方法, 等待服务器端confirm。实际上是一种串行confirm了。

批量confirm模式: 每发送一批消息后, 调用waitForConfirmsOrDie()方法, 等待服务器端confirm。

异步confirm模式: 提供一个回调方法, 服务端confirm了一条或者多条消息后Client端会回调这个方法。

同步Confirm

SendConfirmSync.java

```
try {
    // 通过工厂创建连接
    connection = factory.newConnection();
    // 获取通道
    channel = connection.createChannel();
    // 开启confirm确认模式
    channel.confirmSelect();
    // 声明队列
    channel.queueDeclare(QUEUE_NAME, false, false, false, null);
    // 创建消息
    String message = "Hello world!";
    // 将产生的消息放入队列
    channel.basicPublish("", QUEUE_NAME, null, message.getBytes("UTF-8"));
    System.out.println(" [x] Sent '" + message + "'");
    // 确认消息是否发送成功-单条
    if (channel.waitForConfirms())
        System.out.println("消息发送成功!");
    else
        System.out.println("消息发送失败!");
    // 确认消息是否发送成功-多条
    // 直到所有消息都确认, 只要有一个未确认就会IOException
    channel.waitForConfirmsOrDie();
    System.out.println("消息发送成功!");
}
```



以上代码可以看出，使用同步的方式需要等所有的消息发送成功以后才会执行后面代码，只要有一个消息未被确认就会抛出IO异常。解决办法可以使用异步确认。

异步confirm

异步confirm模式的编程实现最复杂，Channel对象提供的 `confirmListener()` 回调方法只包含 `deliveryTag`（当前Channel发出的消息序号），我们需要自己为每一个Channel维护一个 `unconfirm` 的消息序号集合，每publish一条数据，集合中元素加1，每回调一次 `handleAck` 方法，`unconfirm` 集合删掉相应的一条（`multiple=false`）或多条（`multiple=true`）记录。从程序运行效率上看，这个 `unconfirm` 集合最好采用有序集合 `SortedSet` 存储结构。实际上，`waitForConfirms()` 方法也是通过 `SortedSet` 维护消息序号的。

SendConfirmAsync.java

```
package com.xxxx.confirm.async.send;

import com.rabbitmq.client.Channel;
import com.rabbitmq.client.ConfirmListener;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;

import java.io.IOException;
import java.util.Collections;
import java.util.SortedSet;
import java.util.TreeSet;
import java.util.concurrent.TimeoutException;

/**
 * 信道确认模式-异步-生产者
 */
public class Send {

    // 队列名称
    public static final String QUEUE_NAME = "confirm_async";

    public static void main(String[] args) {
        // 定义连接工厂
        ConnectionFactory factory = new ConnectionFactory();
        factory.setPort(5672);
        factory.setHost("192.168.10.100");
        factory.setUsername("shop");
        factory.setPassword("shop");
        factory.setVirtualHost("/shop");

        Connection connection = null;
        Channel channel = null;
        try {
            // 维护信息发送回执deliveryTag
            final SortedSet<Long>
confirmSet=Collections.synchronizedSortedSet(new TreeSet<Long>());
            // 创建连接
            connection = factory.newConnection();
            // 获取通道
            channel = connection.createChannel();
            // 开启confirm确认模式
```



```
channel.confirmSelect();
// 声明队列
channel.queueDeclare(QUEUE_NAME, false, false, false, null);
// 添加channel 监听
channel.addConfirmListener(new ConfirmListener() {
    // 已确认
    @Override
    public void handleAck(long deliveryTag, boolean multiple) throws
IOException {
        // multiple=true已确认多条 false已确认单条
        if (multiple) {
            System.out.println("handleAck--success-->multiple" +
deliveryTag);

            // 清除前 deliveryTag 项标识id
            confirmSet.headSet(deliveryTag + 1L).clear();
        } else {
            System.out.println("handleAck--success-->single" +
deliveryTag);
            confirmSet.remove(deliveryTag);
        }
    }

    // 未确认
    @Override
    public void handleNack(long deliveryTag, boolean multiple)
throws IOException {
        // multiple=true未确认多条 false未确认单条
        if (multiple) {
            System.out.println("handleNack--failed-->multiple-->" +
deliveryTag);

            // 清除前 deliveryTag 项标识id
            confirmSet.headSet(deliveryTag + 1L).clear();
        } else {
            System.out.println("handleNack--failed-->single" +
deliveryTag);
            confirmSet.remove(deliveryTag);
        }
    }
});
// 循环发送消息演示消息确认
while (true) {
    // 创建消息
    String message = "Hello world!";
    // 获取unconfirm的消息序号deliveryTag
    Long seqNo = channel.getNextPublishSeqNo();
    channel.basicPublish("", QUEUE_NAME, null,
message.getBytes("utf-8"));
    // 将消息序号deliveryTag添加至SortedSet
    confirmSet.add(seqNo);
}
} catch (IOException | TimeoutException e) {
    e.printStackTrace();
} finally {
    try {
        // 关闭通道
        if (null != channel && channel.isOpen())
            channel.close();
        // 关闭连接
```



```
        if (null != connection && connection.isOpen())  
            connection.close();  
    } catch (TimeoutException e) {  
        e.printStackTrace();  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
}  
}  
}
```

异步模式的优点就是执行效率高，不需要等待消息执行完，只需要监听消息即可。

Spring集成RabbitMQ

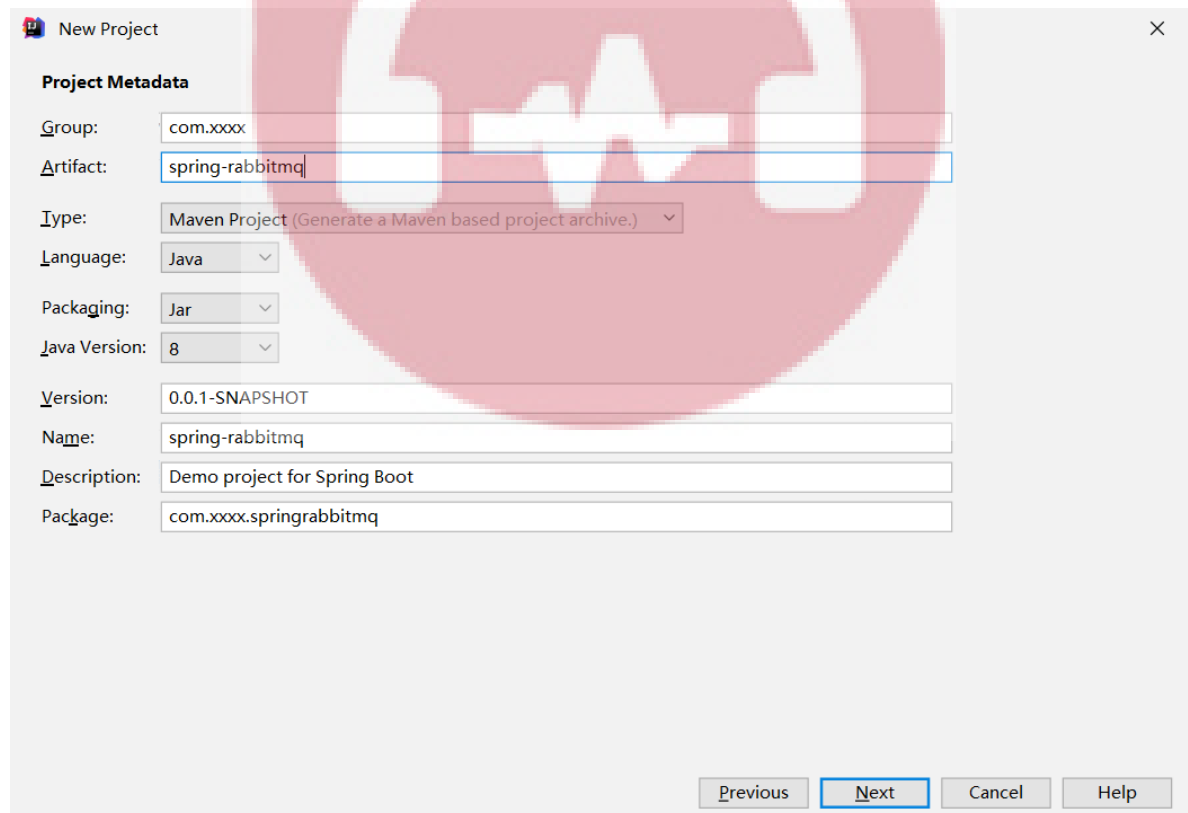
官网: <https://spring.io/projects/spring-amqp>

为什么使用spring AMQP?

基于Spring之上社区活跃
对AMQP协议进行了高度的封装
极大的简化了RabbitMQ的操作
易用性、可扩展

创建聚合项目

创建父项目spring-rabbitmq



New Project

Project Metadata

Group: com.xxxx

Artifact: spring-rabbitmq

Type: Maven Project (Generate a Maven based project archive.)

Language: Java

Packaging: Jar

Java Version: 8

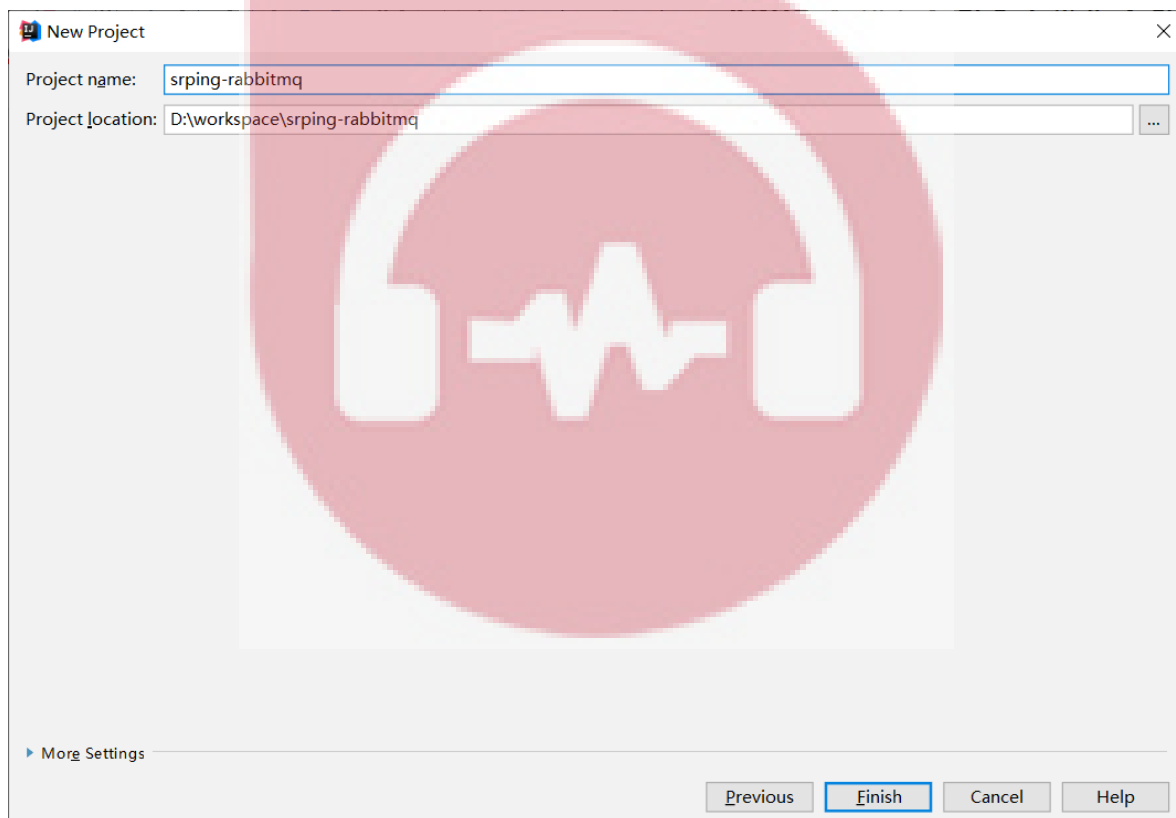
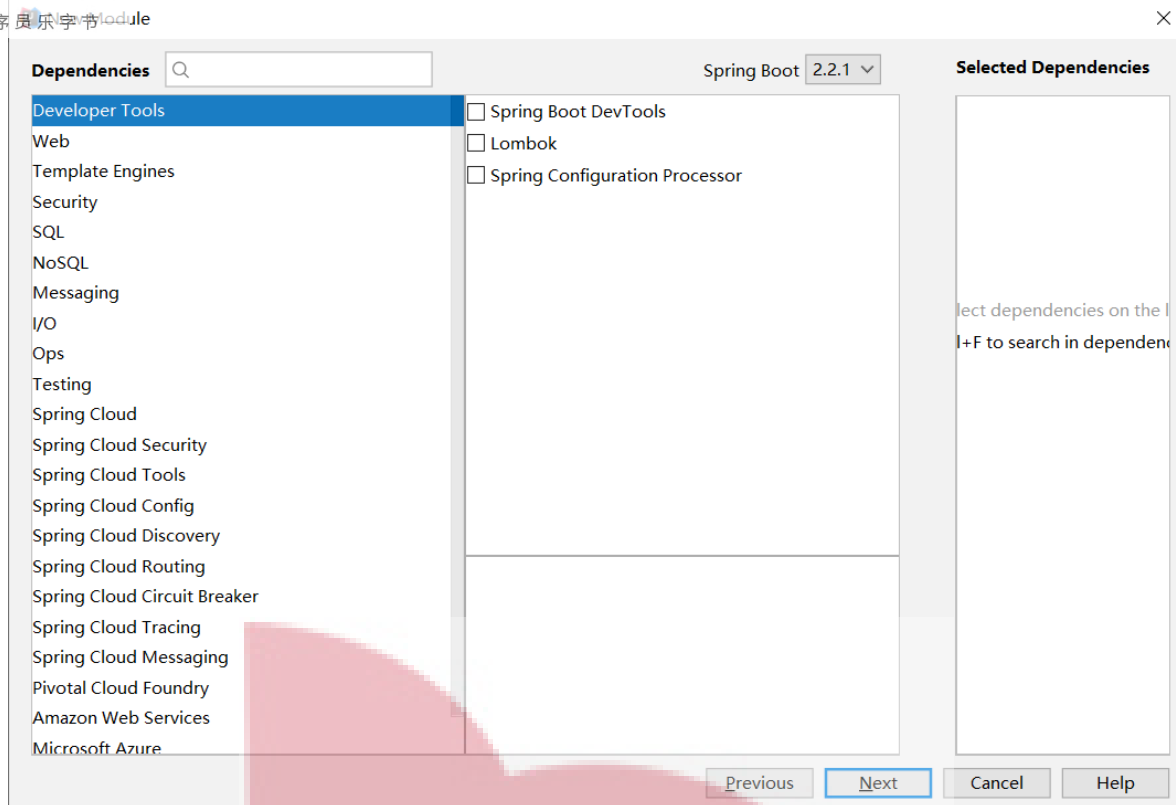
Version: 0.0.1-SNAPSHOT

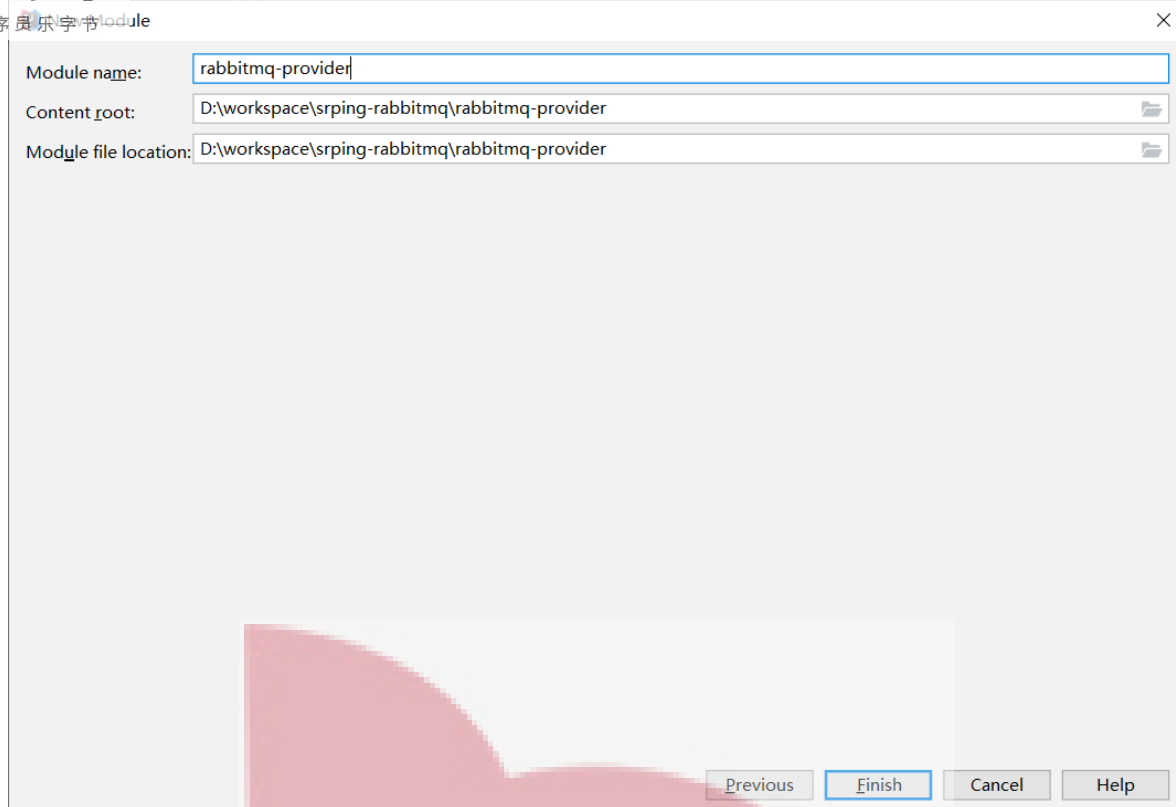
Name: spring-rabbitmq

Description: Demo project for Spring Boot

Package: com.xxxx.springrabbitmq

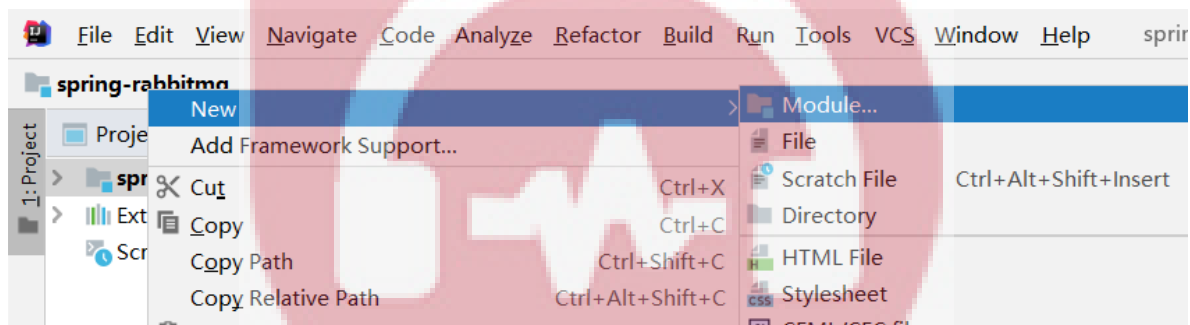
Previous Next Cancel Help





创建rabbitmq-provider

鼠标右键spring-rabbitmq项目new -> Module





Project Metadata

Group:

com.xxxx

Artifact:

rabbit-provider

Type:

Maven Project (Generate a Maven based project archive.)

Language:

Java

Packaging:

Jar

Java Version:

8

Version:

0.0.1-SNAPSHOT

Name:

rabbit-provider

Description:

Demo project for Spring Boot

Package:

com.xxxx.rabbitprovider

Previous

Next

Cancel

Help

New Module

Dependencies

Spring Boot 2.2.1

Developer Tools

Web

Template Engines

Security

SQL

NoSQL

Messaging

I/O

Ops

Testing

Spring Cloud

Spring Cloud Security

Spring Cloud Tools

Spring Cloud Config

Spring Cloud Discovery

Spring Cloud Routing

Spring Cloud Circuit Breaker

Spring Cloud Tracing

Spring Cloud Messaging

Pivotal Cloud Foundry

Amazon Web Services

Microsoft Azure

☐ Spring Integration

☒ Spring for RabbitMQ

☐ Spring for Apache Kafka

☐ Spring for Apache Kafka Streams

☐ Spring for Apache ActiveMQ 5

☐ Spring for Apache ActiveMQ Artemis

☐ WebSocket

☐ RSocket

☐ Apache Camel

☐ Solace PubSub+

Spring for RabbitMQ

Gives your applications a common platform to send and receive messages, and your messages a safe place to live until received.

[Messaging with RabbitMQ](#)

[Reference doc](#)

Selected Dependencies

Web

Spring Web

Messaging

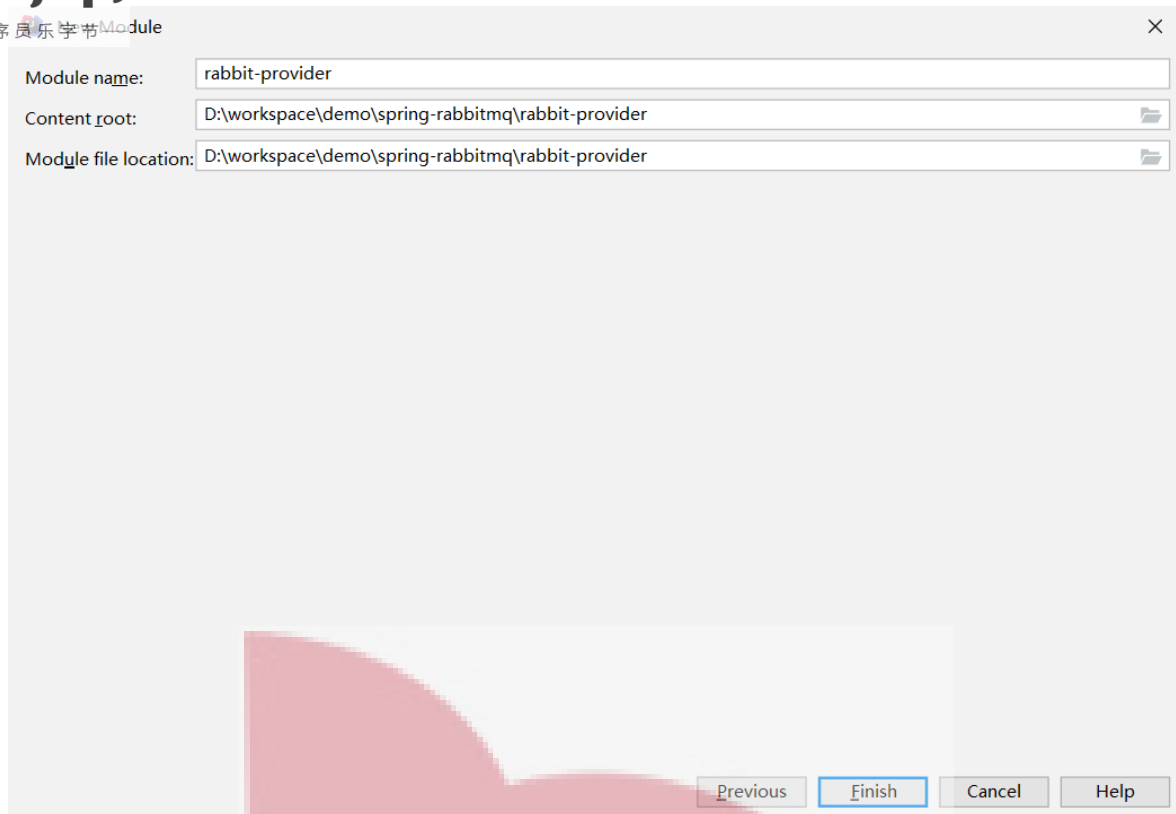
Spring for RabbitMQ

Previous

Next

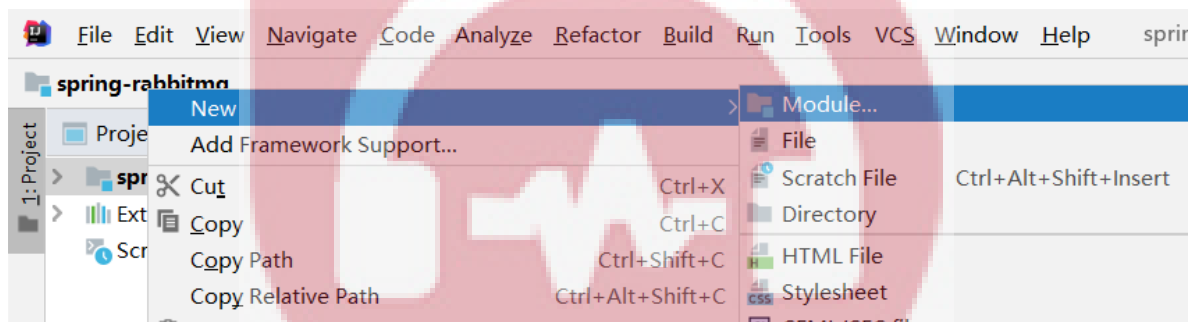
Cancel

Help



创建rabbitmq-consumer

鼠标右键spring-rabbitmq项目new -> Module





Project Metadata

Group:

Artifact:

Type:

Language:

Packaging:

Java Version:

Version:

Name:

Description:

Package:

New Module

Dependencies

- Developer Tools
- Web
- Template Engines
- Security
- SQL
- NoSQL
- Messaging**
- I/O
- Ops
- Testing
- Spring Cloud
- Spring Cloud Security
- Spring Cloud Tools
- Spring Cloud Config
- Spring Cloud Discovery
- Spring Cloud Routing
- Spring Cloud Circuit Breaker
- Spring Cloud Tracing
- Spring Cloud Messaging
- Pivotal Cloud Foundry
- Amazon Web Services
- Microsoft Azure

☐ Spring Integration

☒ **Spring for RabbitMQ**

☐ Spring for Apache Kafka

☐ Spring for Apache Kafka Streams

☐ Spring for Apache ActiveMQ 5

☐ Spring for Apache ActiveMQ Artemis

☐ WebSocket

☐ RSocket

☐ Apache Camel

☐ Solace PubSub+

Spring for RabbitMQ

Gives your applications a common platform to send and receive messages, and your messages a safe place to live until received.

[Messaging with RabbitMQ](#)

[Reference doc](#)

Selected Dependencies

Web

Spring Web

Messaging

Spring for RabbitMQ



Module

Module name: rabbit-consumer

Content root: D:\workspace\demo\spring-rabbitmq\rabbit-consumer

Module file location: D:\workspace\demo\spring-rabbitmq\rabbit-consumer

Previous Finish Cancel Help

父项目依赖

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
      https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.7.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>

  <modules>
    <module>rabbitmq-provider</module>
    <module>rabbitmq-consumer</module>
  </modules>

  <groupId>com.xxxx</groupId>
  <artifactId>spring-rabbitmq</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>pom</packaging>
  <name>spring-rabbitmq</name>
  <description>Demo project for Spring Boot</description>

  <properties>
    <java.version>1.8</java.version>
  </properties>

</project>
```



pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>com.xxxx</groupId>
    <artifactId>spring-rabbitmq</artifactId>
    <version>0.0.1-SNAPSHOT</version>
  </parent>
  <groupId>com.xxxx</groupId>
  <artifactId>rabbitmq-provider</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>rabbitmq-provider</name>
  <description>Demo project for Spring Boot</description>

  <properties>
    <java.version>1.8</java.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-amqp</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
    </dependency>
  </dependencies>
</project>
```

application.yml

```
spring:
  rabbitmq:
    host: 127.0.0.1
    port: 5672
    username: shop
    password: shop
    virtual-host: /shop
server:
  port: 8081
```

RabbitmqConfig.java

```
package com.xxxx.rabbitmqprovider.config;
```




```
import org.springframework.amqp.core.Binding;
import org.springframework.amqp.core.BindingBuilder;
import org.springframework.amqp.core.Queue;
import org.springframework.amqp.core.TopicExchange;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class RabbitmqConfig {
    /**
     * 申明队列
     * @return
     */
    @Bean
    public Queue queue(){
        return new Queue("topics");
    }

    /**
     * 申明交换机（主题模式）
     * @return
     */
    @Bean
    public TopicExchange topicExchange(){
        return new TopicExchange("topicExchange");
    }

    /**
     * 将队列绑定到交换机
     * @return
     */
    @Bean
    public Binding binding(){
        return
        BindingBuilder.bind(queue()).to(topicExchange()).with("topic.msg");
    }
}
```

Send.java

```
package com.xxxx.rabbitmqprovider.send;

import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class Sender {
    @Autowired
    private RabbitTemplate rabbitTemplate;

    public void send() {
        String message = "Hello World!";
        /**
         * 第一个参数：交换机名称
         * 第二个参数：路由key名称
         */
    }
}
```



```
    * 第三个参数：发送的消息
    */
    rabbitTemplate.convertAndSend("topicExchange", "topic.msg", message);
    System.out.println("发送: " + message);
}
}
```

RabbitmqProviderTestApplication.java

```
package com.xxxx.rabbitmqprovider;

import com.xxxx.rabbitmqprovider.send.Sender;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@SpringBootTest(classes = RabbitmqProviderApplication.class)
public class {
    @Autowired
    private Sender sender;

    @Test
    public void testSend(){
        sender.send();
    }
}
```

编写消费者

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>com.xxxx</groupId>
        <artifactId>spring-rabbitmq</artifactId>
        <version>0.0.1-SNAPSHOT</version>
    </parent>

    <groupId>com.xxxx</groupId>
    <artifactId>rabbitmq-consumer</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>rabbitmq-consumer</name>
    <description>Demo project for Spring Boot</description>

    <properties>
        <java.version>1.8</java.version>
    </properties>
```



```
<dependencies>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-test</artifactId>
</dependency>
</dependencies>
</project>
```

application.yml

```
spring:
  rabbitmq:
    host: 127.0.0.1
    port: 5672
    username: shop
    password: shop
    virtual-host: /shop
server:
  port: 8082
```

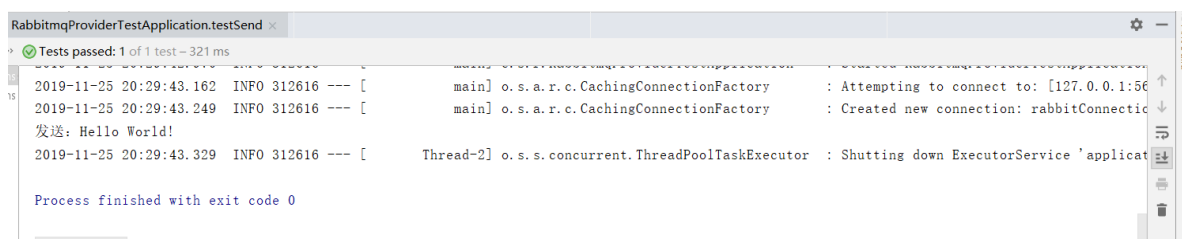
Consumer.java

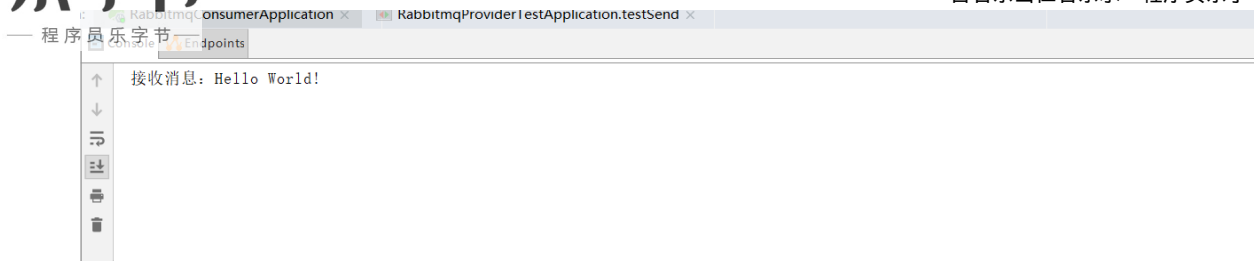
```
package com.xxxx.rabbitmqconsumer.revc;

import org.springframework.amqp.rabbit.annotation.RabbitHandler;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.stereotype.Component;

@Component
//监听队列
@RabbitListener(queues = "topics")
public class Consumer {
    //表示接收消息后的处理方法
    @RabbitHandler
    public void recv(String message){
        System.out.println("接收消息: "+message);
    }
}
```

测试





总结

当然这是官网最简单的例子，以后如果项目是基于配置来做的话要掌握以下：

1. pom中引用jar
2. 先配置rabbitmq的配置
 1. 先配置ConnectionFactory
 2. 配置RabbitAdmin
3. 配置RabbitTemplate这里通常在配置一个Message Convert使用JSON进行数据格式的传输
4. 配置Exchange
5. 配置Queue
6. 配置一个消息处理的bean或者通过Spring扫描，这个Bean最后继承MessageListener 来处理JSON数据
7. 配置Listener Container

