RabbitMQ

目录

RabbitMQ....1
1. 【熟悉】RabbitMQ 概述....3
1.1. 生活中的案例[生产中的问题]为什么要使用 MQ....3
1.2. 为什么要使用 MQ....3
1.3. RabbitMQ 概述及注意点....5
1.4. Rabbitmq 的特点....5
2. 【掌握】RabbitMQ 安装....7
2.1. 安装前的说明....7
2.2. 使用 docker 安装....7
3. 【掌握】RabbitMQ 名词解释....10
3.1. 图说基本概念....10
3.2. 名词解释....10
4. 【熟悉】web 管理界面介绍....11
4.1. overview 概览....11
4.2. Admin 用户和虚拟主机管理....12
4.3. 创建 v-sxt 的虚拟主机和 sxt 用户并分配权限....14
5. 【掌握】RabbitMQ 说明及准备工作....17
5.1. AMQP 协议的回顾....17
5.2. RabbitMQ 支持的消息模型....17
5.3. 创建空项目 rabbitmq-code....19
5.4. 创建 rabbitmq-hello 项目....20
5.5. 修改 pom.xml 引入依赖....20
6. 【掌握】第一种模型(直连)....21
6.1. 概述....21
6.2. 创建生产者并发送测试....21
6.3. 创建消费者并发送测试 Consumer....22
6.4. 有啥用？....24
6.5. 优化工具类....24
6.6. 修改生产者....26
6.7. 修改消费者....27
6.8. 相关参数详解的演示....28
7. 【掌握】第二种模型(work quene)....29
7.1. 概述....29
7.2. 创建生产者....29
7.3. 创建消费者 1....30
7.4. 创建消费者 2....31
7.5. 测试....32

7.6. 消息的自动确认机制【重点】 32
8. 【掌握】第三种模型(fanout) 33
8.1. 概述 33
8.2. 创建生产者 34
8.3. 创建消费者 1 34
8.4. 创建消费者 2 35
8.5. 测试 36
9. 【掌握】第四种模型(Routing)-Direct 36
9.1. 概述 36
9.2. 创建生产者 37
9.3. 创建消费者 1 38
9.4. 创建消费者 2 39
9.5. 测试 40
10. 【掌握】第五种模型(Routing)-Topic 40
10.1. 概述 40
10.2. 创建生产者 40
10.3. 创建消费者 1 41
10.4. 创建消费者 2 42
10.5. 测试 43
11. 【掌握】SpringBoot 中使用 RabbitMQ 43
11.1. 概述 43
11.2. 生产者准备工作 43
11.3. 消费者准备工作 46
11.4. 第一种模型(直连) 49
11.5. 第二种模型(Workquene) 50
11.6. 第三种模型(fanout)广播 52
11.7. 第四种模型(routing-Direct) 55
11.8. 第五种模型(routing-Topic) 58
12. 【掌握】boot 中发送消息的监视 61
12.1. 概述及准备 61
12.2. 消息到交互机和未到达队列里面的监视 62
12.3. 测试 63
12.4. 优化消息监视和到达队列的监视 64
12.5. 消息转换参数的问题 65
13. 【掌握】SpringBoot 中消息的消费及签收机制 65
13.1. 消息的签收机制说明 65
13.2. 自动签收 66
13.3. 手动签收 66
13.4. 不签收 73
13.5. 不去签收消息的死循环怎么解决 73
14. 【掌握】消息的重复消费及处理 78
14.1. 重复消费概述 78
14.2. 利用 BloomFilter 实现去重的操作 78
15. 【掌握】延迟消息+死信消息（常考） 82

15.1. 什么是延迟消息及使用场景.....82
15.2. 对于 Activemq 实现延迟消息【可以回看雷哥的 ActiveMQ】.....83
15.3. Rabbitmq 的延时消息实现.....84

1.【熟悉】RabbitMQ 概述

1.1. 生活中的案例[生产中的问题]为什么要使用 MQ

1，学生问问题的例子
2，分布式项目中 RPC 的调用处理时间过长的问题

1.2. 为什么要使用 MQ

微服务架构后，链式调用是我们在写程序时候的一般流程，为了这完成一个整体功能会把它拆分成多个函数（或子模块）比如模块 A 调用模块 B，模块 B 调用模块 C，模块 C 调用模块 D。但是大型分布式应用中，系统间的 RPC 交互复杂，一个功能后面要调用上百个接口并非不可能，从单机架构过渡到分布式微服务架构，这样的架构有没有问题呢？有

根据上面的几个问题，在设置系统时可以明确要达到的目标

1，要做到系统解耦，当新的模块进来时，可以做到代码改动最小； **能够解耦**
2，设置流程缓冲池，可以让后端系统按自身吞吐能力进行消费，不被冲垮；**能够削峰**
3，强弱依赖梳理能把非关键调用链路的操作异步化并提升整体系统的吞吐能力；**能够异步**

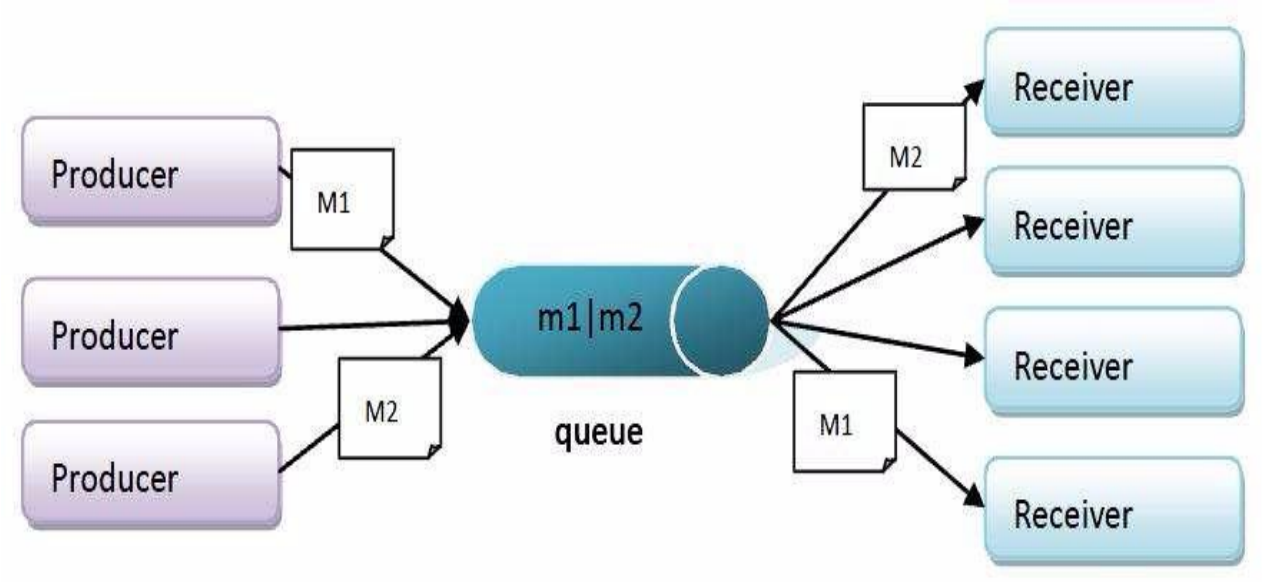
什么是 MQ

1.2.1. 定义

面向消息的中间件(message-oriented middleware0) MOM 能够很好的解决以上的问题。
是指利用高效可靠的消息传递机制进行与平台无关的数据交流，并基于数据通信来进行分布式系统的集成。通过提供消息传递和消息排队模型在分布式环境下提供应用解耦，弹性伸缩，冗余存储，流量削峰，异步通信，数据同步等

大致流程

发送者把消息发给消息服务器，消息服务器把消息存放在若干队列/主题中，在合适的时候，消息服务器会把消息转发给接受者。在这个过程中，发送和接受是异步的，也就是发送无需等待，发送者和接受者的生命周期也没有必然关系在发布 pub/订阅 sub 模式下，也可以完成一对多的通信，可以让一个消息有多个接受者[微信订阅号就是这样的]



1.2.2. 特点

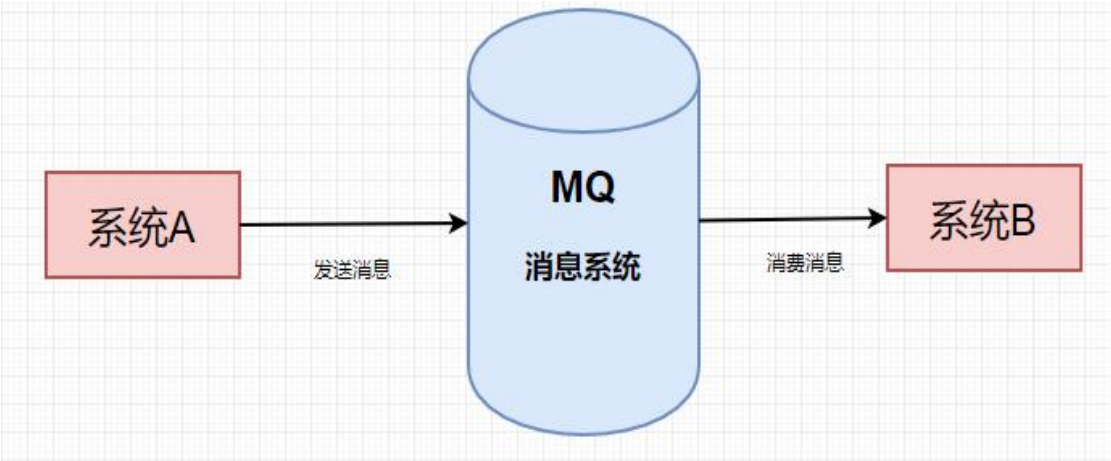
1.2.2.1. 异步处理模式

消息发送者可以发送一个消息而无需等待响应。消息发送者把消息发送到一条虚拟的通道(主题或队列)上;

消息接收者则订阅或监听该通道。一条信息可能最终转发给一个或多个消息接收者，这些接收者都无需对消息发送者做出回应。整个过程都是异步的。

案例:

也就是说，一个系统和另一个系统这间进行通信的时候，假如系统 A 希望发送一个消息给系统 B，让它去处理，但是系统 A 不关注系统 B 到底怎么处理或者有没有处理好，所以系统 A 把消息发送给 MQ，然后就不管这条消息的“死活” 了，接着系统 B 从 MQ 里面消费出来处理即可。至于怎么处理，是否处理完毕，什么时候处理，都是系统 B 的事，与系统 A 无关。

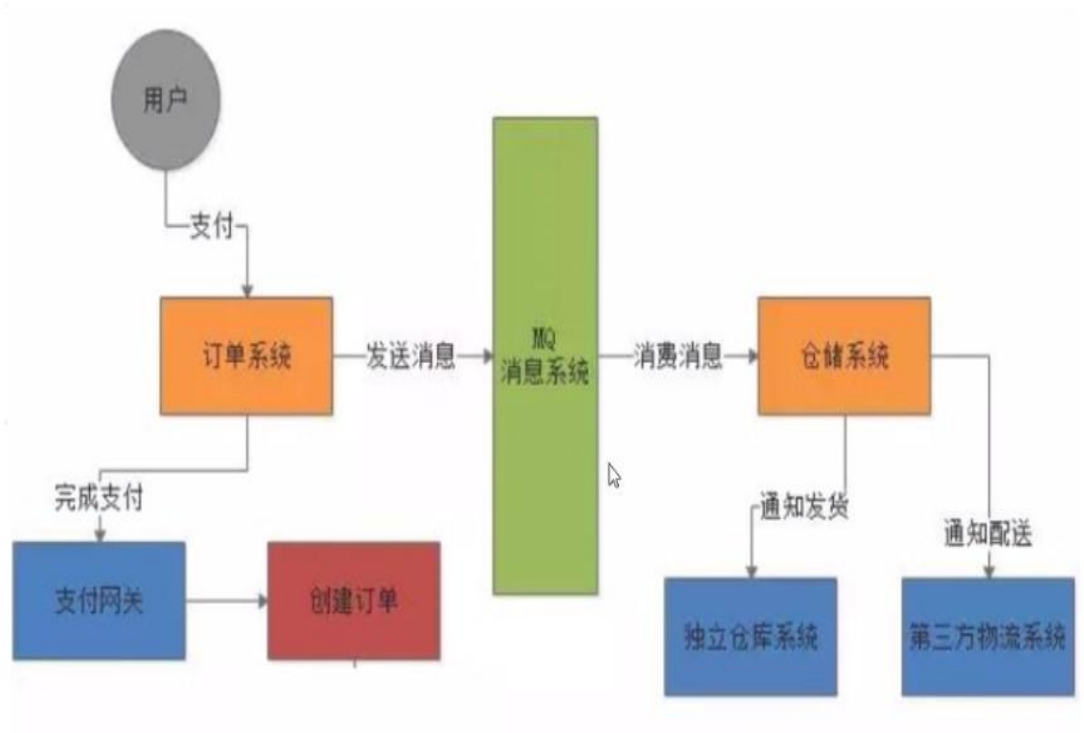


这样的一种通信方式，就是所谓的“异步”通信方式，对于系统 A 来说，只要把消息发给 MQ,然后系统 B 就会异步处去进行处理了，系统 A 不能“同步”的等待系统 B 处理完。这样的好处是什么呢？解耦

1.2.2.2. 应用系统的解耦

发送者和接收者不必了解对方，只需要确认消息
发送者和接收者不必同时在线

1.2.2.3. 现实中的业务



1.3. RabbitMQ 概述及注意点

RabbitMQ 是实现了高级消息队列协议（AMQP）的开源消息代理软件（亦称面向消息的中间件）。RabbitMQ 服务器是用 Erlang 语言编写的，而集群和故障转移是构建在开放电信平台框架上的。所有主要的编程语言均有与代理接口通讯的客户端库。

总结：RabbitMQ 实现了 AMQP 协议来构建自己的消息队列
　　RabbitMQ 是 Erlang 语言写的，但是我们有操作 RabbitMQ 的 java 的驱动

面试问题： Activemq 和 RabbitMq 的区别

根本的区别：Activemq 他实现的是 JMS 协议（Java 消息协议）
　　RabbitMQ 实现的是 AMQP 协议（高级消息队列协议）

Activemq ： 是 Java 写的

RabbitMQ ： Erlang 写的： 吞吐更多，延时更低！！

当然：区别还有很大，你一学就知道了

1.4. Rabbitmq 的特点

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

1.4.1. 可靠性（Reliability）

RabbitMQ 使用一些机制来保证可靠性，如持久化、传输确认、发布确认。

1.4.2. 灵活的路由（Flexible Routing）

在消息进入队列之前，通过 Exchange 来路由消息的。对于典型的路由功能，RabbitMQ 已经提供了一些内置的 Exchange 来实现。针对更复杂的路由功能，可以将多个 Exchange 绑定在一起，也通过插件机制实现自己的 Exchange 。

1.4.3. 消息集群（Clustering）

多个 RabbitMQ 服务器可以组成一个集群，形成一个逻辑 Broker 。

1.4.4. 高可用（Highly Available Queues）

队列可以在集群中的机器上进行镜像，使得在部分节点出问题的情况下队列仍然可用。

1.4.5. 多种协议（Multi-protocol）

RabbitMQ 支持多种消息队列协议，比如 STOMP、MQTT 等等。

1.4.6. 多语言客户端（Many Clients）

RabbitMQ 几乎支持所有常用语言，比如 Java、.NET、Ruby 等等。

1.4.7. 管理界面（Management UI）

RabbitMQ 提供了一个易用的用户界面，使得用户可以监控和管理消息 Broker 的许多方面。

1.4.8. 跟踪机制（Tracing）

如果消息异常，RabbitMQ 提供了消息跟踪机制，使用者可以找出发生了什么。

1.4.9. 插件机制（Plugin System）

RabbitMQ 提供了许多插件，来从多方面进行扩展，也可以编写自己的插件。

2.【掌握】RabbitMQ 安装

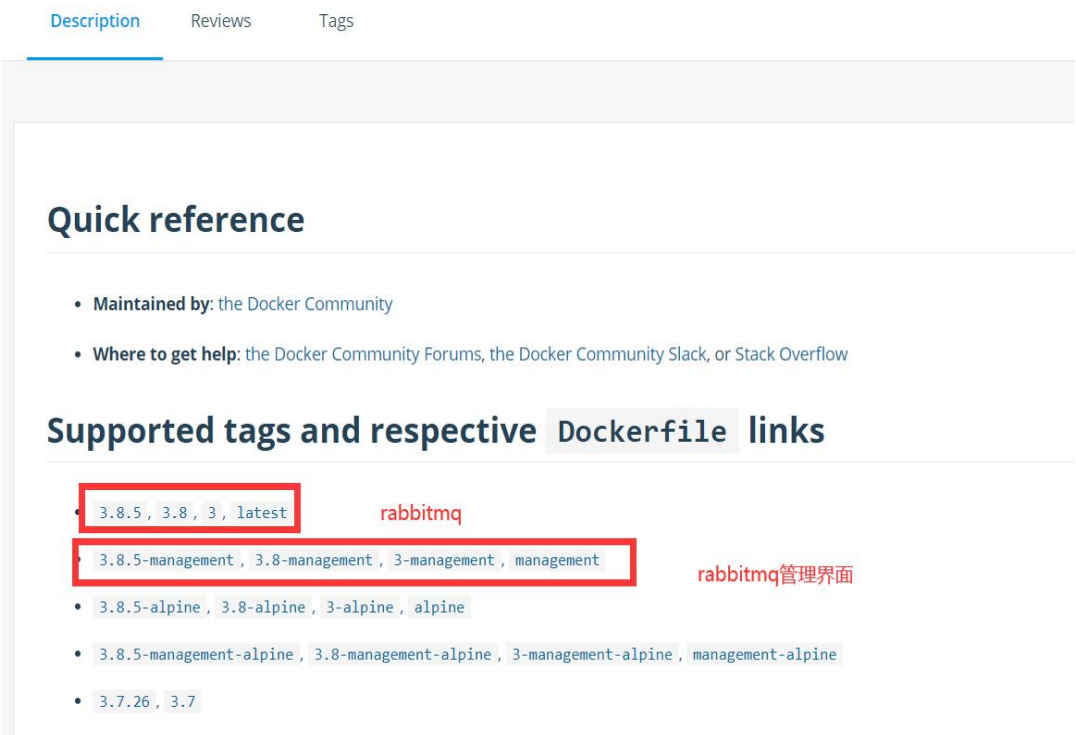
2.1. 安装前的说明

Rabbitmq->Erlang -> 安装 Erlang 虚拟机->跑 Rabbitmq 比较麻烦
Rabbitmq 对 docker 的支持非常到位！官网经常更新镜像，所以怎么办呢。废话，用 docker 跑啊

2.2. 使用 docker 安装

2.2.1. 下载镜像

https://hub.docker.com/_/rabbitmq




```
docker pull rabbitmq
```

```
[root@iZbp183x79rt6tw0lv5795Z ~]# docker pull rabbitmq
Using default tag: latest
Trying to pull repository docker.io/library/rabbitmq ...
latest: Pulling from docker.io/library/rabbitmq
d7c3167c320d: Already exists
131f805ec7fd: Already exists
322ed380e680: Already exists
6ac240b13098: Already exists
58ab633708c7: Pull complete
4ef7b4c52e3f: Extracting [===========>                                    ] 7.209 MB/32.42 MB
0bcc8241708b: Download complete
4bbf89f47f34: Download complete
2dcee968b577: Download complete
b7f34b3a6ae9: Download complete
```

```
docker.io/rabbitmq                          latest              7e50c60f7e3e        5 days ago
```

Rabbitmq 里面也有控制台界面，但是它们不是一起的，你需要控制台的，需要下载另一个镜像

docker pull rabbitmq:management

```
[root@iZbp183x79rt6tw0lv5795Z ~]# docker pull rabbitmq:management
Trying to pull repository docker.io/library/rabbitmq ...
management: Pulling from docker.io/library/rabbitmq
d7c3167c320d: Already exists
131f805ec7fd: Already exists
322ed380e680: Already exists
6ac240b13098: Already exists
58ab633708c7: Already exists
4ef7b4c52e3f: Already exists
0bcc8241708b: Already exists
4bbf89f47f34: Already exists
2dcee968b577: Already exists
b7f34b3a6ae9: Already exists
57f11a70a594: Pull complete
6a2415c21710: Pull complete
Digest: sha256:e1620d50e92d53d548829c881a39cc547240333b276f600ebfa75d02d5a87e38
Status: Downloaded newer image for docker.io/rabbitmq:management
[root@iZbp183x79rt6tw0lv5795Z ~]#
```

2.2.2. 运行容器

15672：图形化界面的端口

5672 ：数据的端口

docker run --name rabbitmq -p 15672:15672 -p 5672:5672 -e RABBITMQ_DEFAULT_USER=user -e RABBITMQ_DEFAULT_PASS=123456 -d rabbitmq:management

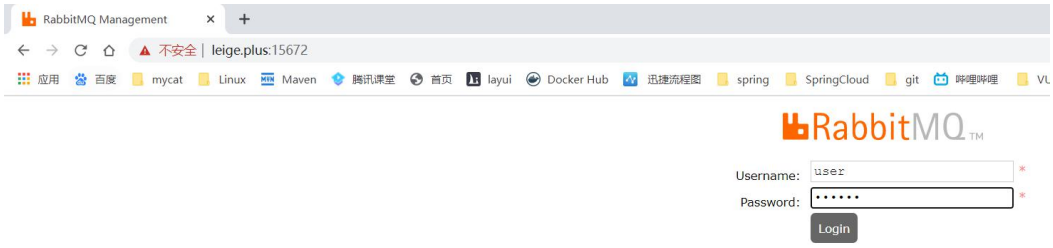
2.2.3. 开发阿里云端口

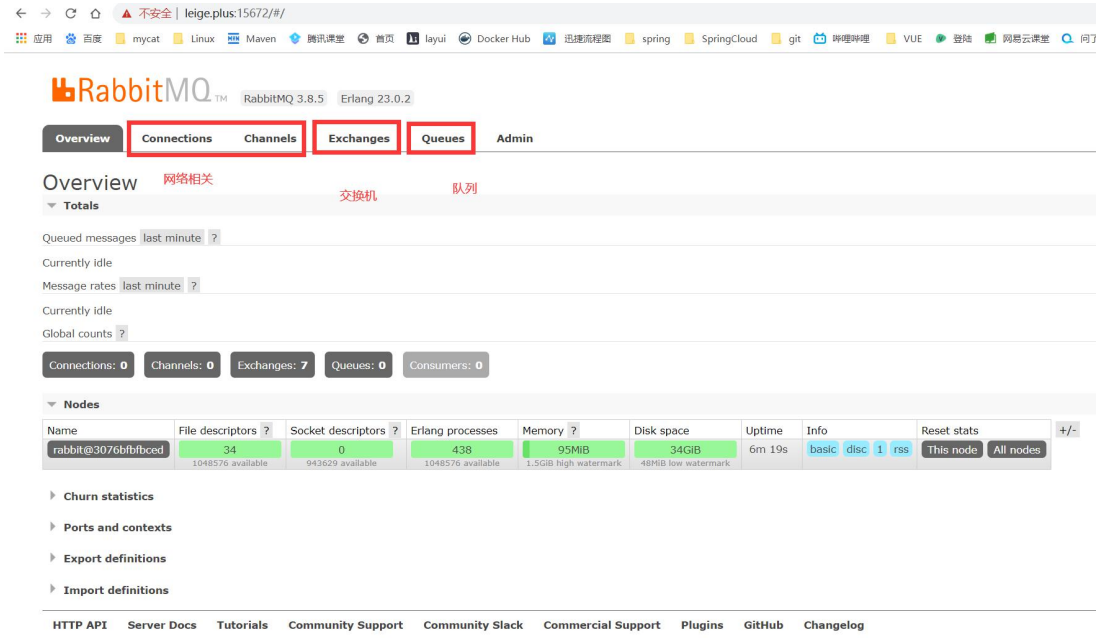
手动添加 快速添加 全部编辑

授权策略	优先级	协议类型	端口范围
允许	1	自定义 TCP	目的:15672/15672
允许	1	自定义 TCP	目的:5672/5672

2.2.4. 访问 rabbitMQ

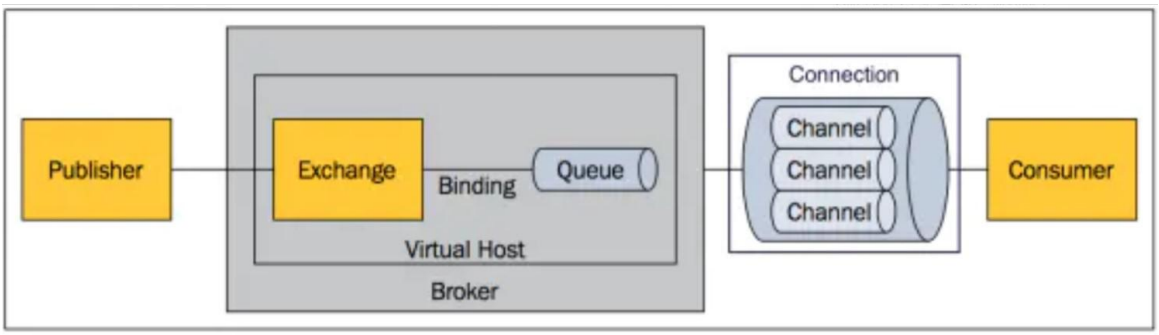
ip:15672





3.【掌握】RabbitMQ 名词解释

3.1. 图说基本概念



3.2. 名词解释

3.2.1. Message

消息，消息是不具名的，它由消息头和消息体组成。消息体是不透明的，而消息头则由一系列的可选属性组成，这些属性包括 routing-key（路由键）、priority（相对于其他消息的优先权）、delivery-mode（指出该消息可能需要持久性存储）等。

3.2.2. Publisher

消息的生产者，也是一个向交换器发布消息的客户端应用程序。

3.2.3. Exchange

交换器，用来接收生产者发送的消息并将这些消息路由给服务器中的队列。

3.2.4. Binding

绑定，用于消息队列和交换器之间的关联。一个绑定就是基于路由键将交换器和消息队列连接起来的路由规则，所以可以将交换器理解成一个由绑定构成的路由表。

3.2.5. Queue

消息队列，用来保存消息到发送给消费者。它是消息的容器，也是消息的终点。一个消息可投入一个或多

个队列。消息一直在队列里面，等待消费者连接到这个队列将其取走。

3.2.6. Connection

网络连接，比如一个 TCP 连接。

3.2.7. Channel

信道，多路复用连接中的一条独立的双向数据流通道。信道是建立在真实的 TCP 连接内地虚拟连接，AMQP 命令都是通过信道发出去的，不管是发布消息、订阅队列还是接收消息，这些动作都是通过信道完成。因为对于操作系统来说建立和销毁 TCP 都是非常昂贵的开销，所以引入了信道的概念，以复用一条 TCP 连接。

3.2.8. Consumer

消息的消费者，表示一个从消息队列中取得消息的客户端应用程序。

3.2.9. Virtual Host

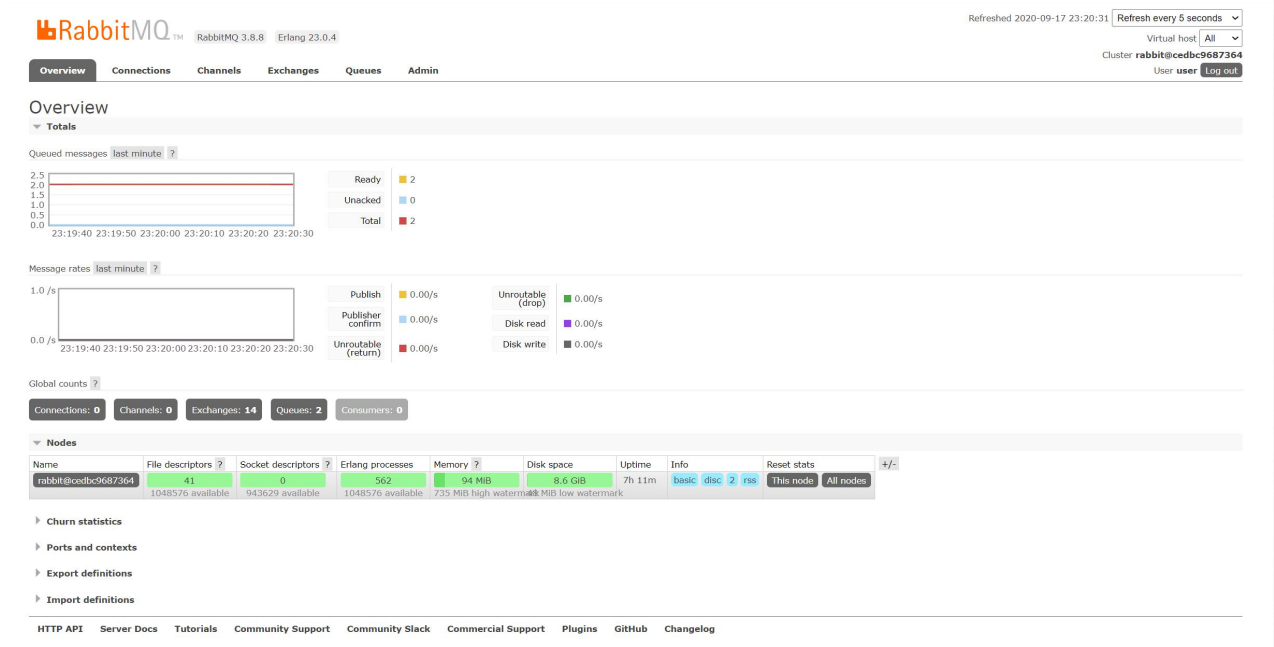
虚拟主机，表示一批交换器、消息队列和相关对象。虚拟主机是共享相同的身份认证和加密环境的独立服务器域。每个 vhost 本质上就是一个 mini 版的 RabbitMQ 服务器，拥有自己的队列、交换器、绑定和权限机制。vhost 是 AMQP 概念的基础，必须在连接时指定，RabbitMQ 默认的 vhost 是 / 。

3.2.10. Broker

表示消息队列服务器实体

4.【熟悉】web 管理界面介绍

4.1. overview 概览



connections：无论生产者还是消费者，都需要与 RabbitMQ 建立连接后才可以完成消息的生产和消费，在这里可以查看连接情况

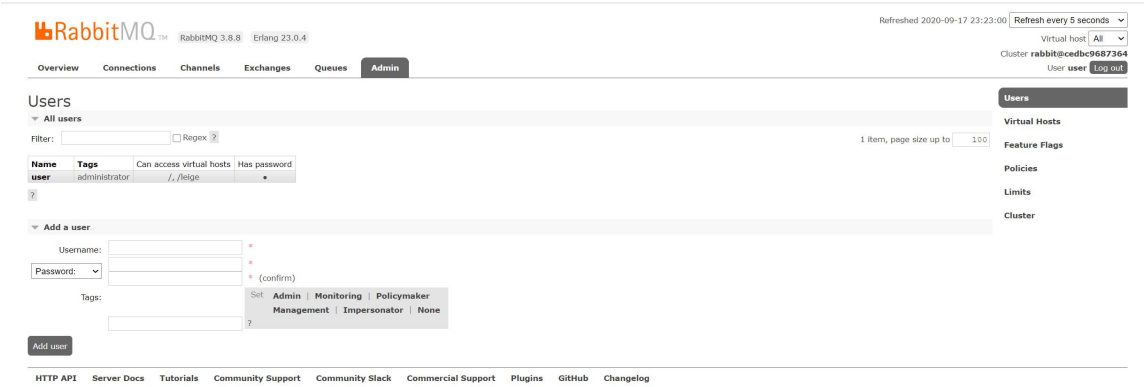
channels：通道，建立连接后，会形成通道，消息的投递获取依赖通道。

Exchanges：交换机，用来实现消息的路由

Queues：队列，即消息队列，消息存放在队列中，等待消费，消费后被移除队列。

4.2. Admin 用户和虚拟主机管理

4.2.1. 添加用户



上面的 Tags 选项，其实是指定用户的角色，可选的有以下几个：

超级管理员(administrator)

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

监控者(monitoring)

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

策略制定者(policymaker)

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

普通管理者(management)

仅可登陆管理控制台，无法看到节点信息，也无法对策略进行管理。

其他

无法登陆管理控制台，通常就是普通的生产者和消费者。

4.2.2. 创建虚拟主机

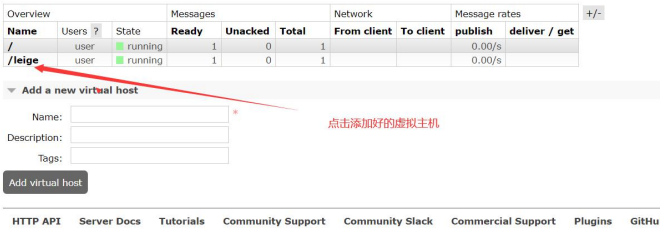
虚拟主机是什么

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

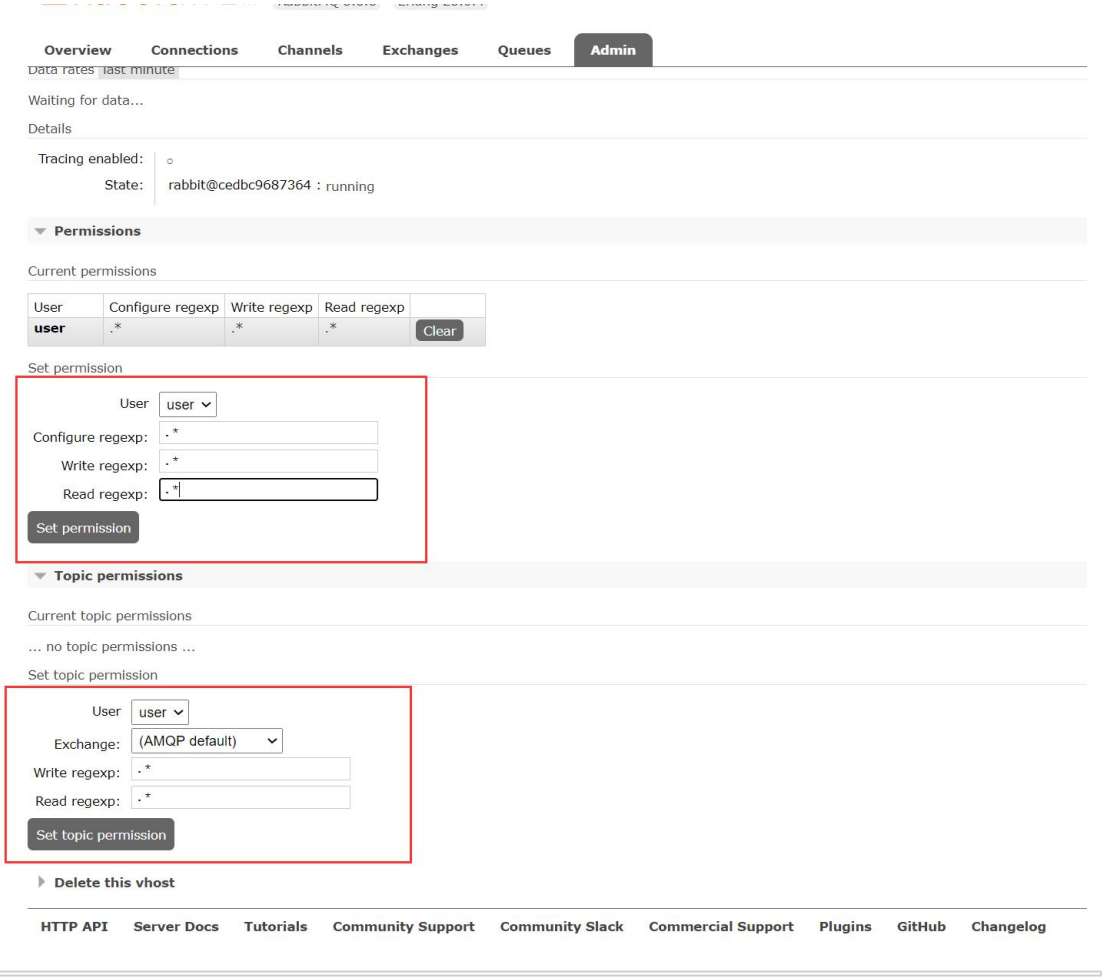
4.2.3. 绑定虚拟主机和用户

创建好虚拟主机，我们还要给用户添加访问权限：

点击添加好的虚拟主机：

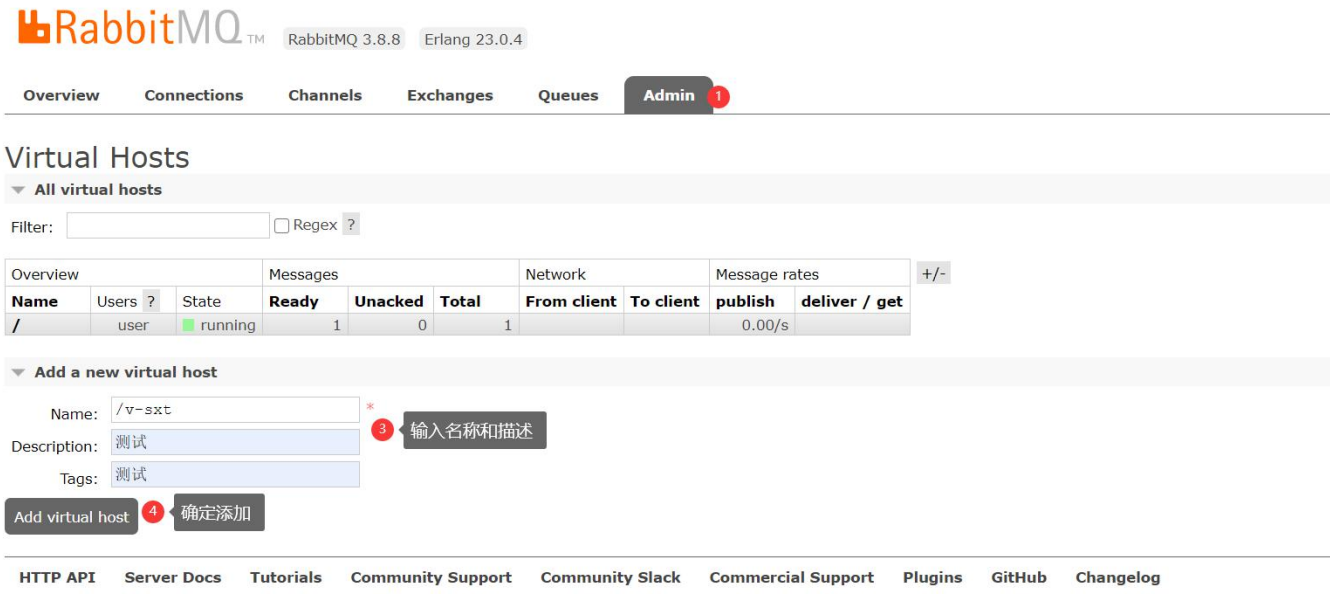


进入虚拟机设置界面：



4.3. 创建 v-sxt 的虚拟主机和 sxt 用户并分配权限

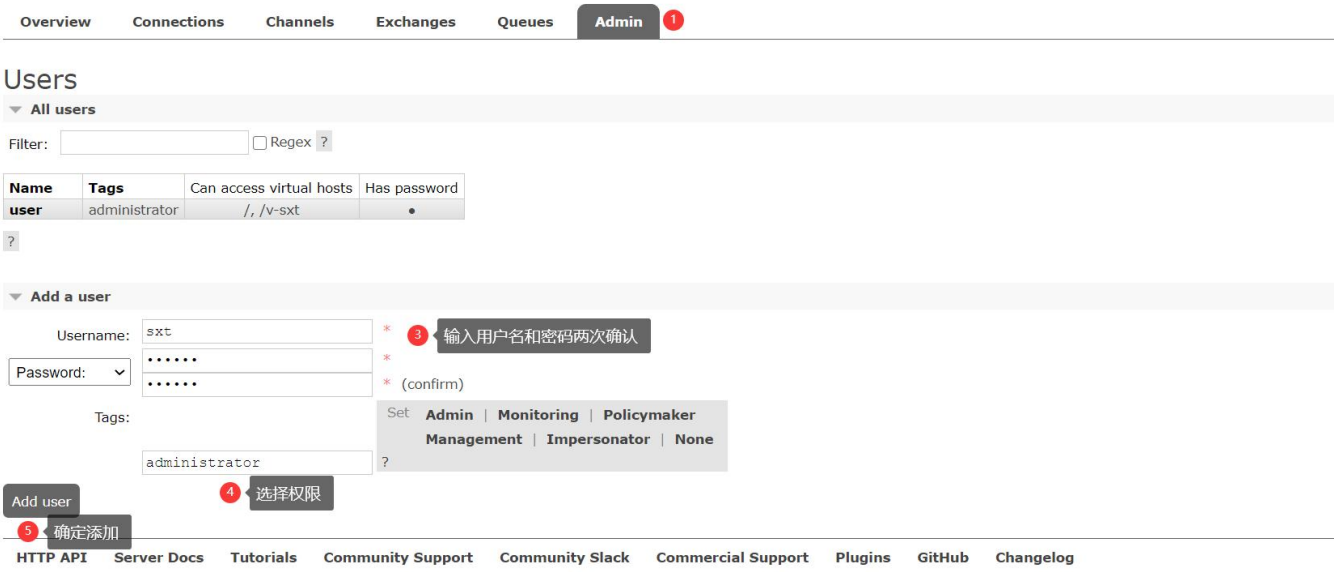
4.3.1. 创建 v-sxt 虚拟主机



添加成功

Overview			Messages			Network		Message rates		+/-
Name	Users ?	State	**Ready**	**Unacked**	**Total**	**From client**	**To client**	**publish**	**deliver / get**	
/	user	running	1	0	1			0.00/s		
/v-sxt	user	running	NaN	NaN	NaN					

4.3.2. 创建 sxt 用户

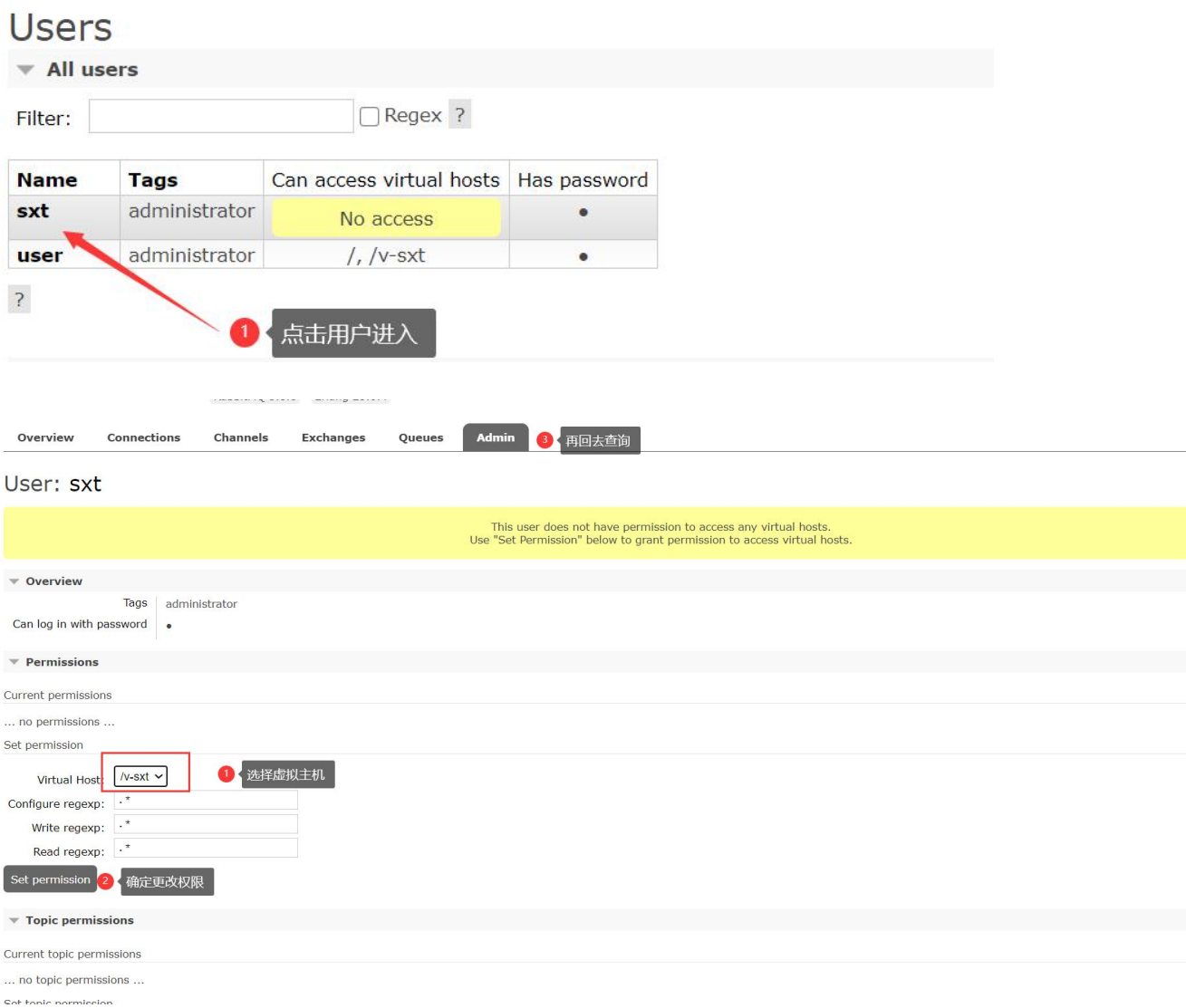


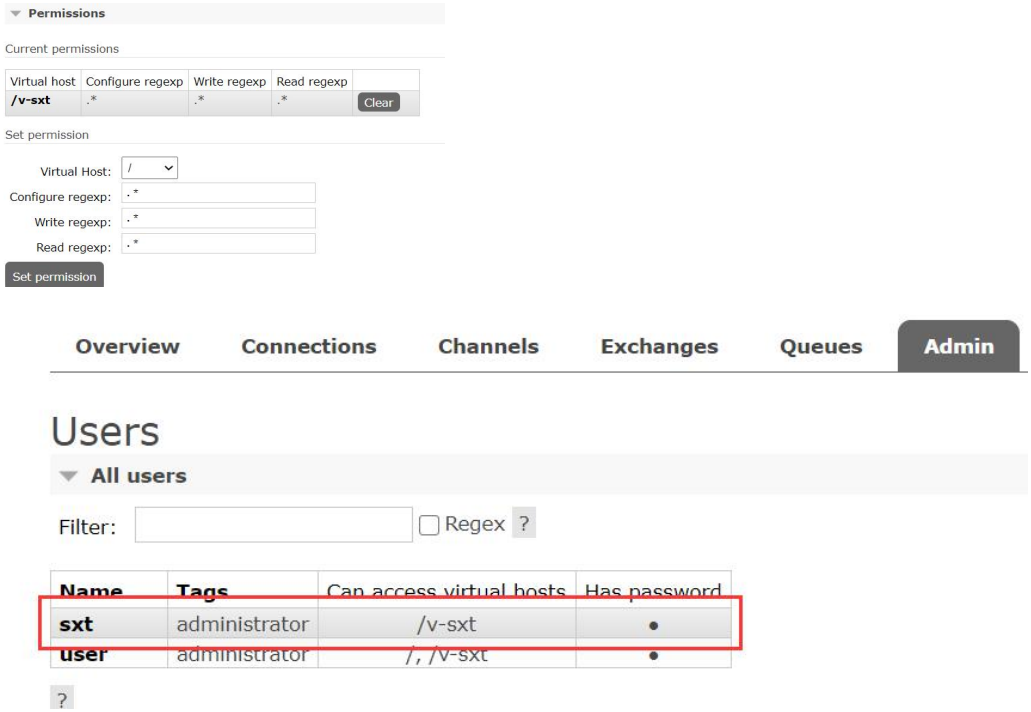
添加成功，但是现在不能访问任何虚拟主机

Name	Tags	Can access virtual hosts	Has password
sxt	administrator	No access	•
user	administrator	/, /v-sxt	•

?

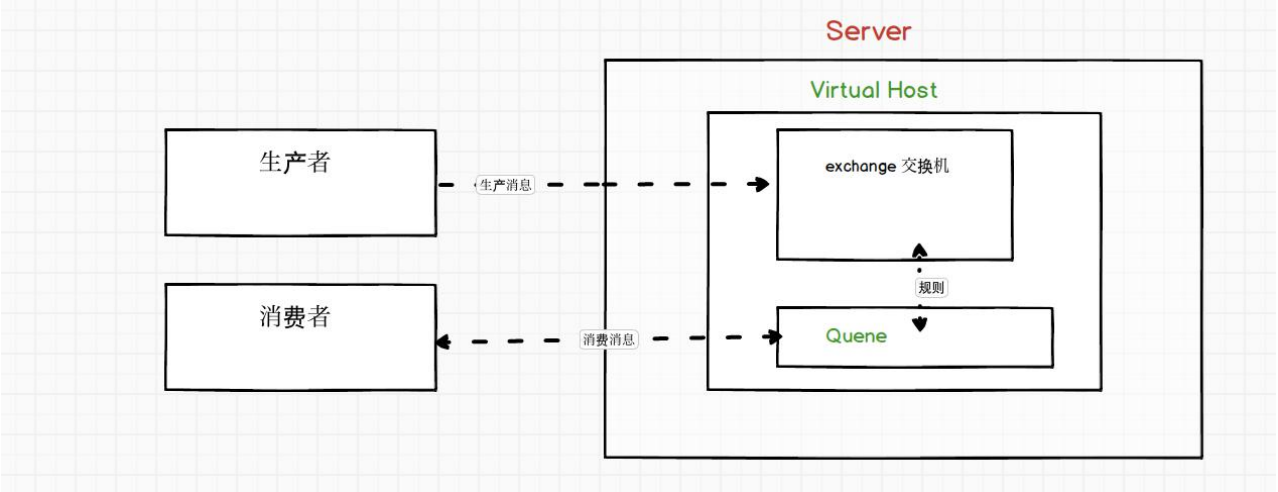
4.3.3. 分配权限





5.【掌握】RabbitMQ 说明及准备工作

5.1. AMQP 协议的回顾



5.2. RabbitMQ 支持的消息模型

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

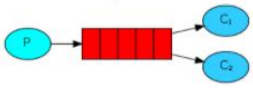
1 **"Hello World!"**

The simplest thing that does *something*

- Python
- Java
- Ruby
- PHP
- C#
- JavaScript
- Go
- Elixir
- Objective-C
- Swift
- Spring AMQP

2 **Work queues**

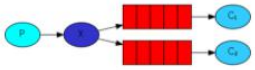
Distributing tasks among workers (the competing consumers pattern)



- Python
- Java
- Ruby
- PHP
- C#
- JavaScript
- Go
- Elixir
- Objective-C
- Swift
- Spring AMQP

3 **Publish/Subscribe**

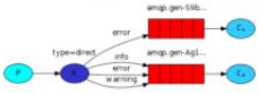
Sending messages to many consumers at once



- Python
- Java
- Ruby
- PHP
- C#
- JavaScript
- Go
- Elixir
- Objective-C
- Swift
- Spring AMQP

4 **Routing**

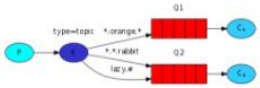
Receiving messages selectively



- Python
- Java
- Ruby
- PHP
- C#
- JavaScript
- Go
- Elixir
- Objective-C
- Swift
- Spring AMQP

5 **Topics**

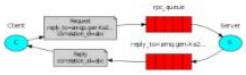
Receiving messages based on a pattern (topics)



- Python
- Java
- Ruby
- PHP
- C#
- JavaScript
- Go
- Elixir
- Objective-C
- Swift
- Spring AMQP

6 **RPC**

Request/reply pattern example



- Python
- Java
- Ruby
- PHP
- C#
- JavaScript
- Go
- Elixir
- Spring AMQP

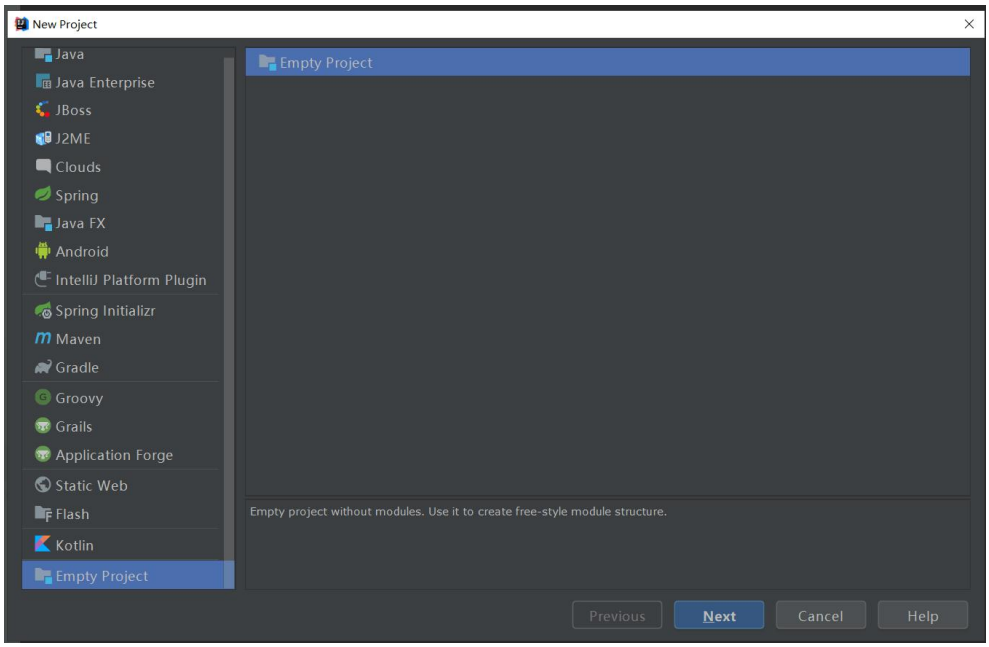
7 Publisher Confirms

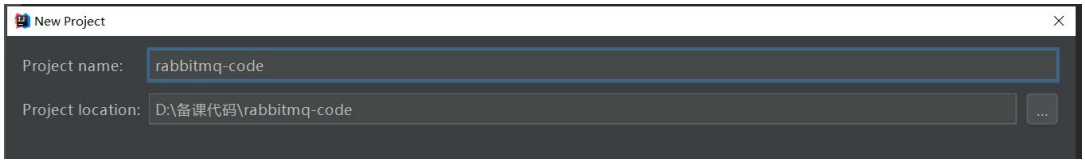
Reliable publishing with publisher confirms

- Java
- C#

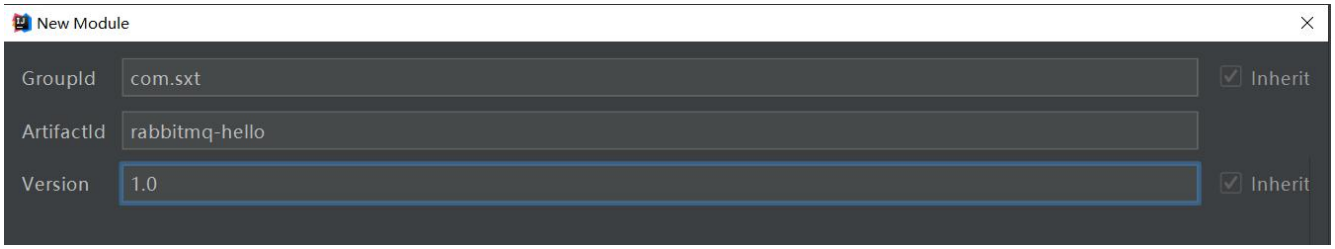
注意 3.7 的版本不支持第 7 种模式

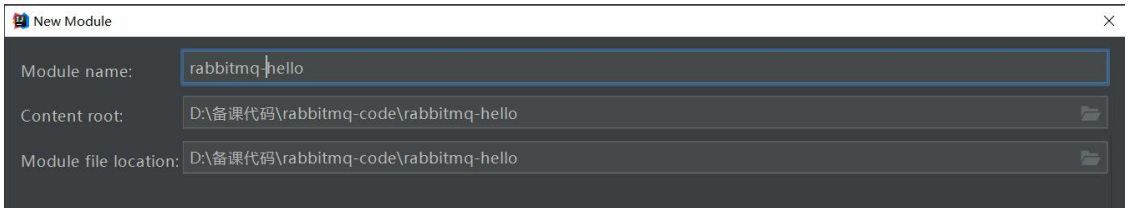
5.3. 创建空项目 rabbitmq-code





5.4. 创建 rabbitmq-hello 项目





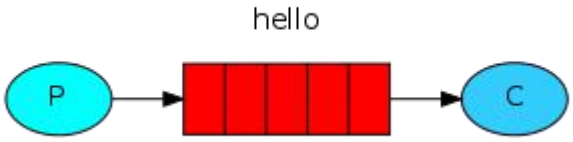
5.5. 修改 pom.xml 引入依赖

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
```

```
  </dependency>
  <dependency>
    <groupId>com.rabbitmq</groupId>
    <artifactId>amqp-client</artifactId>
    <version>5.9.0</version>
  </dependency>
</dependencies>
```

6.【掌握】第一种模型(直连)

6.1. 概述



在上图的模型中，有以下概念：

P：生产者，也就是要发送消息的程序

C：消费者：消息的接受者，会一直等待消息到来。

queue：消息队列，图中红色部分。类似一个邮箱，可以缓存消息；生产者向其中投递消息，消费者从其中取出消息。

6.2. 创建生产者并发送测试

```
package a_direct;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;
import org.junit.Test;
import java.io.IOException;
import java.util.concurrent.TimeoutException;
/**
 * @Author: 尚学堂 雷哥
 * 消息生产者
 */
public class Producer {
    @Test
    public void sendMessage() throws IOException, TimeoutException {
```

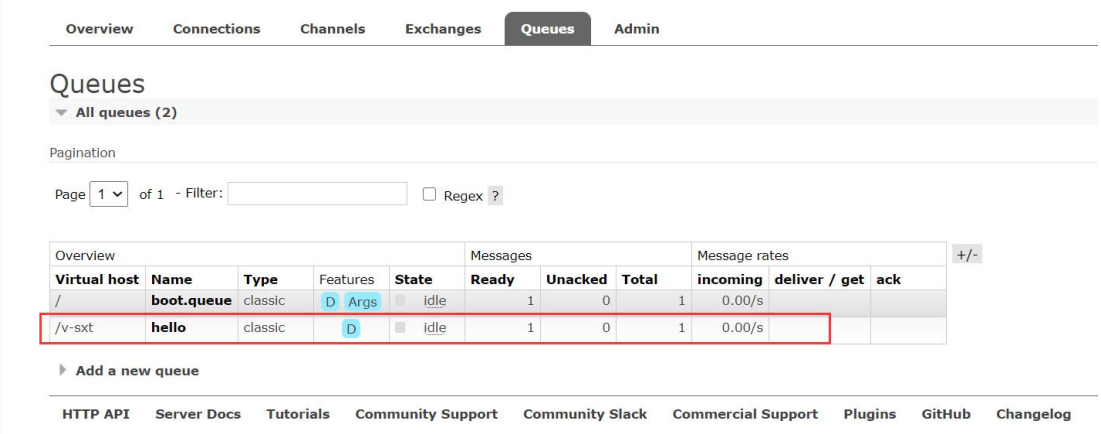
```
        //创建连接工厂
        ConnectionFactory connectionFactory=new ConnectionFactory();
        //设置相关参数
        connectionFactory.setHost("116.62.44.5");
        connectionFactory.setPort(5672);
        connectionFactory.setUsername("sxt");
        connectionFactory.setPassword("123456");
        connectionFactory.setVirtualHost("/v-sxt");
        //从连接工厂里面创建一个连接
        Connection connection = connectionFactory.newConnection();
        //创建通道
        Channel channel = connection.createChannel();
        /**
         * 绑定队列
         * 参数 1：队列名　如果发送消息时，队列在 mq 里面不存在，它会创建一个新的队列
         * 参数 2：是否持久化 如果为 false 当 MQ 重启时，消息会丢失
         * 参数 3：是否独享。true 代表只有当前的 connection 可以访问这个队列
         * 参数 4：是否自动删除　是否用完之后就删除
         * 参数 5：其它参数
         */
        channel.queueDeclare("hello",true,false,false,null);

        /**
         * 发送消息
         * 参数 1 交换机名称　暂时用不到
         * 参数 2　队列名
         * 参数 3　基础参数　是否持久
         * 参数 4　消息的内容
         */
        channel.basicPublish("","hello",null,"hello rabbitmq".getBytes());

        //关闭通道和连接
        channel.close();
        connection.close();
        System.out.println("消息发送成功");
    }
}
```

6.3. 创建消费者并发送测试 Consumer

```
package a_direct;
```

```
import com.rabbitmq.client.*;
import org.junit.Test;

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

/**
 * @Author: 尚学堂 雷哥
 * 消息消费者
 */
public class Consumer {

    @Test
    public void receiveMessage() throws IOException, TimeoutException {
        //创建连接工厂
        ConnectionFactory connectionFactory=new ConnectionFactory();
        //设置相关参数
        connectionFactory.setHost("116.62.44.5");
        connectionFactory.setPort(5672);
        connectionFactory.setUsername("sxt");
        connectionFactory.setPassword("123456");
        connectionFactory.setVirtualHost("/v-sxt");
        //从连接工厂里面创建一个连接
        Connection connection = connectionFactory.newConnection();
        //创建通道
        Channel channel = connection.createChannel();
        /**
         * 绑定队列
         * 参数 1：队列名  如果发送消息时，队列在 mq 里面不存在，它会创建一个新的队列
         * 参数 2：是否持久化 如果为 false 当 MQ 重启时，消息会丢失
         * 参数 3：是否独享。true 代表只有当前的 connection 可以访问这个队列
         * 参数 4：是否自动删除  是否用完之后就删除
         * 参数 5：其它参数
         */
        channel.queueDeclare("hello",true,false,false,null);

        /**
         * 接收消息
         * 参数 1：队列名
         * 参数 2：是否自动签收
         * 参数 3：回调接口
         *
         */
        channel.basicConsume("hello",true,new DefaultConsumer(channel){
```

```
            /**
             * @param consumerTag
             * @param envelope
             * @param properties
             * @param body   消息体
             * @throws IOException
             */
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties
properties, byte[] body) throws IOException {
                System.out.println("消费者收到消息："+new String(body));
            }
        });

        //不能让程序结束
        System.in.read();
        //关闭
//          channel.close();
//          connection.close();
//          System.out.println("消息消费成功");
    }
}
```



6.4. 有啥用?

这种就是一个点对点的发送和消费，一个生产者，一个消费者，可以用于登陆发短信等功能

6.5. 优化工具类

```
package utils;
```

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

/**
 * @Author: 尚学堂 雷哥
 * 工具类
 */
public class RabbitMQUtils {

    private static ConnectionFactory connectionFactory;
    static{
        //创建连接工厂
        connectionFactory=new ConnectionFactory();
        //设置相关参数
        connectionFactory.setHost("116.62.44.5");
        connectionFactory.setPort(5672);
        connectionFactory.setUsername("sxt");
        connectionFactory.setPassword("123456");
        connectionFactory.setVirtualHost("/v-sxt");
    }

    /**
     * 提供一个获取连接的方法
     */
    public static Connection getConnection(){
        try {
            //从连接工厂里面创建一个连接
            Connection connection = connectionFactory.newConnection();
            return connection;
        }catch (Exception e){
            e.printStackTrace();
        }
        return null;
    }

    /**
     * 提供一个可以关闭通道和连接的方法
     */
    public static void closeChannelAndConnection(Channel channel,Connection connection){
        try{
            if(null!=channel)channel.close();
            if(null!=connection) connection.close();
```

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

6.6. 修改生产者

```
package a_direct;

import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;
import org.junit.Test;
import utils.RabbitMQUtils;
import java.io.IOException;
import java.util.concurrent.TimeoutException;
/**
 * @Author: 尚学堂 雷哥
 * 消息生产者
 */
public class Producer {
    @Test
    public void sendMessage() throws IOException, TimeoutException {

        //从连接工厂里面创建一个连接
        Connection connection = RabbitMQUtils.getConnection();
        //创建通道
        Channel channel = connection.createChannel();
        /**
         * 绑定队列
         * 参数 1：队列名　如果发送消息时，队列在 mq 里面不存在，它会创建一个新的队列
         * 参数 2：是否持久化 如果为 false 当 MQ 重启时，消息会丢失
         * 参数 3：是否独享。true 代表只有当前的 connection 可以访问这个队列
         * 参数 4：是否自动删除　是否用完之后就删除
         * 参数 5：其它参数
         */
        channel.queueDeclare("hello",true,false,false,null);

        /**
         * 发送消息
         * 参数 1 交换机名称　暂时用不到
```

```
            * 参数 2   队列名
            * 参数 3   基础参数   是否持久
            * 参数 4   消息的内容
            */
            channel.basicPublish("","hello",null,"hello rabbitmq".getBytes());

            //关闭通道和连接
            RabbitMQUtils.closeChannelAndConnection(channel,connection);
            System.out.println("消息发送成功");
        }
}
```

6.7. 修改消费者

```
package a_direct;
import com.rabbitmq.client.*;
import org.junit.Test;
import utils.RabbitMQUtils;
import java.io.IOException;
import java.util.concurrent.TimeoutException;
/**
 * @Author: 尚学堂 雷哥
 * 消息消费者
 */
public class Consumer {
    @Test
    public void receiveMessage() throws IOException, TimeoutException {
        //从连接工厂里面创建一个连接
        Connection connection = RabbitMQUtils.getConnection();
        //创建通道
        Channel channel = connection.createChannel();
        /**
         * 绑定队列
         * 参数 1：队列名   如果发送消息时，队列在 mq 里面不存在，它会创建一个新的队列
         * 参数 2：是否持久化 如果为 false 当 MQ 重启时，消息会丢失
         * 参数 3：是否独享。true 代表只有当前的 connection 可以访问这个队列
         * 参数 4：是否自动删除   是否用完之后就删除
         * 参数 5：其它参数
         */
        channel.queueDeclare("hello",true,false,false,null);
        /**
         * 接收消息
         * 参数 1：队列名
```

```
            * 参数 2：是否自动签收
            * 参数 3：回调接口
            *
            */
        channel.basicConsume("hello",true,new DefaultConsumer(channel){
            /**
             * @param consumerTag
             * @param envelope
             * @param properties
             * @param body   消息体
             * @throws IOException
             */
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties
properties, byte[] body) throws IOException {
                System.out.println("消费者收到消息："+new String(body));
            }
        });
        //不能让程序结束
        System.in.read();
        //关闭
        RabbitMQUtils.closeChannelAndConnection(channel,connection);
//          channel.close();
//          connection.close();
//          System.out.println("消息消费成功");
    }
}
```

6.8. 相关参数详解的演示

如果只设置队列的持久化，消息默认的是不会持久化的

```
/**
 * 参数1  queue 队列名称，如果不存在，则自动创建
 * 参数2  durable 是否持久化
 * 参数3  exclusive 是否独占队列
 * 参数4  autoDelete 是否自动删除是否在消费完成之自动删除队列 如果长时间不用
 * 参数5  arguments  其他属性
 */
channel.queueDeclare( queue: "hello", durable: true, exclusive: false, autoDelete: false, arguments: null);
/**
```

```
 */
channel.queueDeclare( queue: "hello", durable: true, exclusive: false, autoDelete: false, arguments: null);
/**
 * 发布消息的方法
 * 参数1：交换机：因为现在是直连，所有不用经过交换机
 * 参数2：队列名称
 * 参数3：传递消息的额外设置
 * 参数4：消息的具体内容
 */
channel.basicPublish( exchange: "", routingKey: "hello", MessageProperties.PERSISTENT_TEXT_PLAIN,"hello rabbitmq".getBytes());
```

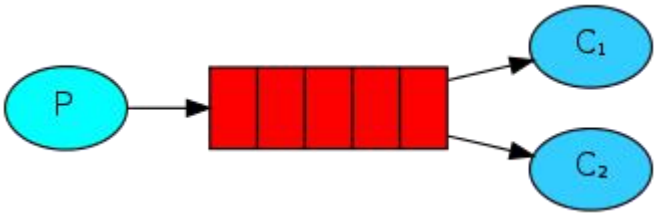
持久化队列

持久化消息

7.【掌握】第二种模型(work quene)

7.1. 概述

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



角色：

P：生产者：任务的发布者

C1：消费者-1，领取任务并且完成任务，假设完成速度较慢

C2：消费者-2：领取任务并完成任务，假设完成速度快

7.2. 创建生产者

```
package b_workqueue;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.MessageProperties;
import org.junit.Test;
import utils.RabbitMQUtils;
import java.io.IOException;
import java.util.concurrent.TimeoutException;
/**
 * @Author: 尚学堂 雷哥
 * 消息生产者
 */
public class Producer {
    @Test
    public void sendMessage() throws IOException, TimeoutException {
        Connection connection = RabbitMQUtils.getConnection();
        Channel channel = connection.createChannel();
        channel.queueDeclare("hello",false,false,false,null);
        for (int i = 1; i <=100 ; i++) {
```

```
                channel.basicPublish("","hello", null,("hello rabbitmq
workqueue--"+i).getBytes());
            }
            //关闭通道和连接
            RabbitMQUtils.closeChannelAndConnection(channel,connection);
            System.out.println("消息发送成功");
        }
}
```

7.3. 创建消费者 1

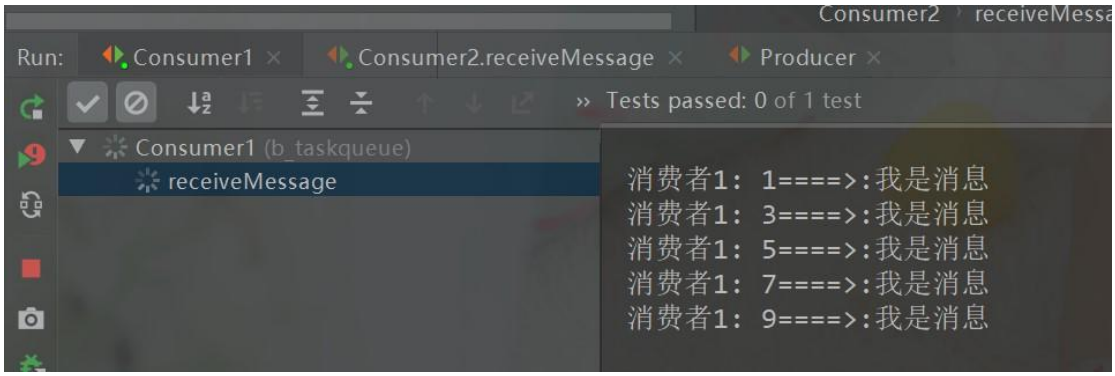
```
package b_workqueue;
import com.rabbitmq.client.*;
import org.junit.Test;
import utils.RabbitMQUtils;
import java.io.IOException;
import java.util.concurrent.TimeoutException;
/**
 * @Author: 尚学堂 雷哥
 * 消息消费者
 */
public class Consumer1 {

    @Test
    public void receiveMessage() throws IOException, TimeoutException {
        Connection connection = RabbitMQUtils.getConnection();
        Channel channel = connection.createChannel();
        channel.queueDeclare("hello",false,false,false,null);
        channel.basicConsume("hello",true,new DefaultConsumer(channel){
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
                System.out.println("消费者【1】收到消息："+new String(body));
            }
        });
        System.out.println("消费者【1】启动成功");
        //不能让程序结束
        System.in.read();
        //关闭
        RabbitMQUtils.closeChannelAndConnection(channel,connection);
    }
}
```

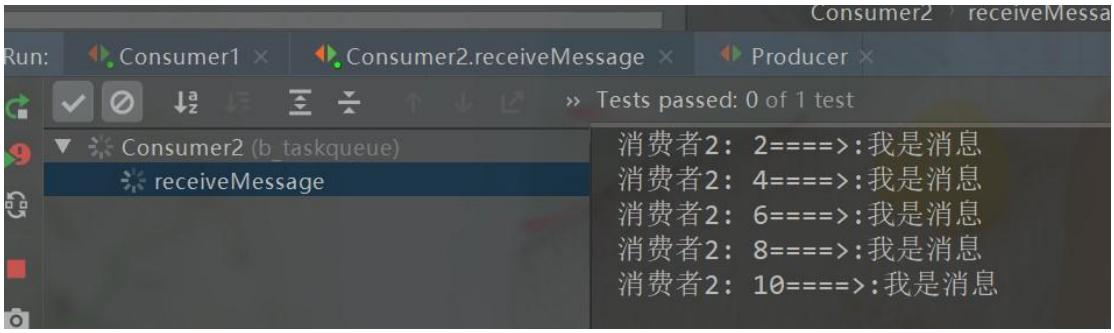
7.4. 创建消费者 2

```
package b_workqueue;

import com.rabbitmq.client.*;
import org.junit.Test;
import utils.RabbitMQUtils;
import java.io.IOException;
import java.util.concurrent.TimeoutException;
/**
 * @Author: 尚学堂 雷哥
 * 消息消费者
 */
public class Consumer2 {
    @Test
    public void receiveMessage() throws IOException, TimeoutException {
        Connection connection = RabbitMQUtils.getConnection();
        Channel channel = connection.createChannel();
//          channel.basicQos(1);//一次只处理一条消息
        channel.queueDeclare("hello",false,false,false,null);
        //把签收模式变成 false
        channel.basicConsume("hello",false,new DefaultConsumer(channel){
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
//                  try {
//                      Thread.sleep(500);
//                  } catch (InterruptedException e) {
//                      e.printStackTrace();
//                  }
                //手动签收
                //channel.basicAck(envelope.getDeliveryTag(),false);
                System.out.println("消费者【2】收到消息："+new String(body));
            }
        });
        System.out.println("消费者【2】启动成功");
        //不能让程序结束
        System.in.read();
        //关闭
        RabbitMQUtils.closeChannelAndConnection(channel,connection);
    }
}
```

7.5. 测试

先启动消费者 1 和消费者 2，再发送消息，发现结果如下





他们是平均消费的 官网有说明 `https://www.rabbitmq.com/tutorials/tutorial-two-java.html`

By default, RabbitMQ will send each message to the next consumer, in sequence. On average every consumer will get the same number of messages. This way of distributing messages is called round-robin. Try this out with three or more workers.

那么实际开发中可能有消费者处理的慢，有的处理的快，那么如何配置呢 引入自动确认机制

7.6. 消息的自动确认机制【重点】

https://www.rabbitmq.com/tutorials/tutorial-two-java.html

Message acknowledgment

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

完成一项任务可能需要几秒钟。您可能想知道，如果一个消费者开始了一项很长的任务，但只完成了一部分就去世了，会发生什么。在我们当前的代码中，一旦 RabbitMQ 向使用者传递了一条消息，它就会立即将其标记为删除。在这种情况下，如果你杀死一个工人，我们就会丢失它正在处理的信息。我们还将丢失所有发送给这个特定 worker 但尚未处理的消息。

但我们不想失去任何任务。如果一个工人死亡，我们希望任务被交付给另一个工人。

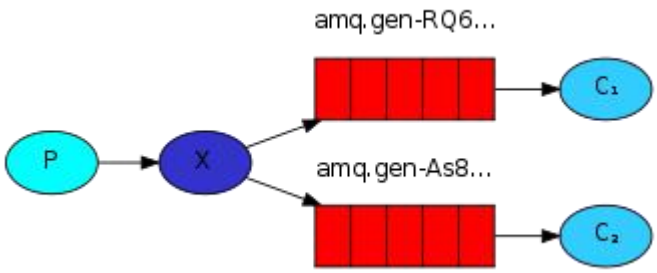
```
//设置一次只接收一条未确认的消息
    channel.basicQos(2);
    //接收消息 参数2:关闭自动确认消息
```

```
    channel.basicConsume("hello",false,new DefaultConsumer(channel){
      @Override
      public void handleDelivery(String consumerTag, Envelope envelope,
BasicProperties properties,
          byte[] body) throws IOException {
//        try {
//          Thread.sleep(500);
//        } catch (InterruptedException e) {
//          e.printStackTrace();
//        }
//        i++;
//        if(i<=2){
//          int a=10/0;
//        }
        System.out.println("消息者【1】接收到消息："+new String(body));
        //手动签收消息
        channel.basicAck(envelope.getDeliveryTag(),false);
      }
    });
```

8.【掌握】第三种模型(fanout)

8.1. 概述

fanout 扇出 也称为广播



在广播模式下，消息发送流程是这样的：

- 可以有多个消费者
- 每个消费者有自己的 queue（队列）
- 每个队列都要绑定到 Exchange（交换机）
- 生产者发送的消息，只能发送到交换机，交换机来决定要发给哪个队列，生产者无法决定。
- 交换机把消息发送给绑定过的所有队列

- 队列的消费者都能拿到消息。实现一条消息被多个消费者消费

8.2. 创建生产者

```
package c_fanout;
import com.rabbitmq.client.BuiltinExchangeType;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import org.junit.Test;
import utils.RabbitMQUtils;
import java.io.IOException;
/**
 * @Author: 尚学堂 雷哥
 * fanout 发送者测试
 */
public class Producer {
    @Test
    public void sendMessage() throws IOException {
        Connection connection = RabbitMQUtils.getConnection();
        //创建通道
        Channel channel = connection.createChannel();
        //设置交换机
//          channel.exchangeDeclare("logs", "fanout");
        channel.exchangeDeclare("logs", BuiltinExchangeType.FANOUT);
        //向交换机发消息
        channel.basicPublish("logs","",null,("我是个 fanout 类型的消息").getBytes());
        RabbitMQUtils.closeChannelAndConnection(channel,connection);
        System.out.println("消费发送成功");
    }
}
```

8.3. 创建消费者 1

```
package c_fanout;
import com.rabbitmq.client.*;
import org.junit.Test;
import utils.RabbitMQUtils;
import java.io.IOException;
/**
 * @Author: 尚学堂 雷哥
 * fanout 类型交换机的消费者类
```

```
 */
public class Consumer1 {
    @Test
    public void receiveMessage() throws IOException {
        Connection connection = RabbitMQUtils.getConnection();
        //得到通道
        Channel channel = connection.createChannel();
        //绑定交换机
        channel.exchangeDeclare("logs", BuiltinExchangeType.FANOUT);
        //从通道里面得到一个临时队列
        String queue = channel.queueDeclare().getQueue();
        //把临时队列和交换机绑定
        channel.queueBind(queue,"logs","");
        //接收消息
        channel.basicConsume(queue,new DefaultConsumer(channel){
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
                System.out.println("消费者【1】接收到消息:"+new String(body));
            }
        });
        System.out.println("消费者【1】启动成功");
        System.in.read();
    }
}
```

8.4. 创建消费者 2

```
package c_fanout;
import com.rabbitmq.client.*;
import org.junit.Test;
import utils.RabbitMQUtils;
import java.io.IOException;
/**
 * @Author: 尚学堂 雷哥
 * fanout 类型交换机的消费者类
 */
public class Consumer2 {
    @Test
    public void receiveMessage() throws IOException {
        Connection connection = RabbitMQUtils.getConnection();
        //得到通道
        Channel channel = connection.createChannel();
```

```
        //绑定交换机
        channel.exchangeDeclare("logs", BuiltinExchangeType.FANOUT);
        //从通道里面得到一个临时队列
        String queue = channel.queueDeclare().getQueue();
        //把临时队列和交换机绑定
        channel.queueBind(queue,"logs","");
        //接收消息
        channel.basicConsume(queue,new DefaultConsumer(channel){
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
                System.out.println("消费者【2】接收到消息:"+new String(body));
            }
        });
        System.out.println("消费者【2】启动成功");
        System.in.read();
    }
}
```

8.5. 测试

先启动两个消费者，再发消息

9.【掌握】第四种模型(Routing)-Direct

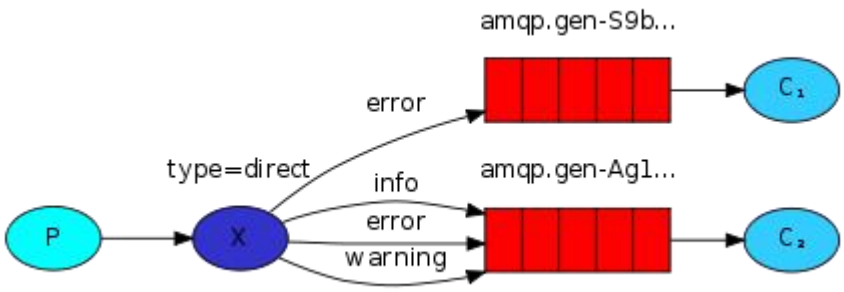
9.1. 概述

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

在 Direct 模型下：

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

流程：



图解：

P：生产者，向 Exchange 发送消息，发送消息时，会指定一个 routing key。

X：Exchange（交换机），接收生产者的消息，然后把消息递交给 与 routing key 完全匹配的队列

C1：消费者，其所在队列指定了需要 routing key 为 error 的消息

C2：消费者，其所在队列指定了需要 routing key 为 info、error、warning 的消息

9.2. 创建生产者

```
package d_routing_direct;

import com.rabbitmq.client.BuiltinExchangeType;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import org.junit.Test;
import utils.RabbitMQUtils;
/**
 * @Author: 尚学堂 雷哥
 *
 * 交换机直连模式
 */
public class Producer {
    @Test
    public void sendMessage() throws  Exception{
        Connection connection = RabbitMQUtils.getConnection();
        Channel channel = connection.createChannel();
        String exchangeName = "logs_direct";
```

```
        channel.exchangeDeclare(exchangeName, BuiltinExchangeType.DIRECT);
        //声明一个路由 key
        String routingKey="error";
        channel.basicPublish(exchangeName,routingKey,null,
                ("我是一个直连类型的交换机消息-routingKey:"+routingKey).getBytes());
        System.out.println("消息发送成功");
        RabbitMQUtils.closeChannelAndConnection(channel,connection);
    }
}
```

9.3. 创建消费者 1

```
package d_routing_direct;
import com.rabbitmq.client.*;
import org.junit.Test;
import utils.RabbitMQUtils;
import java.io.IOException;

/**
 * @Author: 尚学堂 雷哥
 *
 * 交换机直连模式
 */
public class Consumer1 {

    @Test
    public void receiveMessage() throws  Exception{
        Connection connection = RabbitMQUtils.getConnection();
        Channel channel = connection.createChannel();
        String exchangeName = "logs_direct";
        channel.exchangeDeclare(exchangeName, BuiltinExchangeType.DIRECT);
        //得到一个临时队列
        String queue = channel.queueDeclare().getQueue();
        //绑定队列到交换机
        channel.queueBind(queue,exchangeName,"error");
        //消费消息
        channel.basicConsume(queue,new DefaultConsumer(channel){
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
                System.out.println("消费者【1】收到消息:"+new String(body));
            }
        });
```

```
        System.in.read();
    }
}
```

9.4. 创建消费者 2

```
package d_routing_direct;

import com.rabbitmq.client.*;
import org.junit.Test;
import utils.RabbitMQUtils;

import java.io.IOException;

/**
 * @Author: 尚学堂 雷哥
 *
 * 交换机直连模式
 */
public class Consumer2 {

    @Test
    public void receiveMessage() throws   Exception{
        Connection connection = RabbitMQUtils.getConnection();
        Channel channel = connection.createChannel();
        String exchangeName = "logs_direct";
        channel.exchangeDeclare(exchangeName, BuiltinExchangeType.DIRECT);
        //得到一个临时队列
        String queue = channel.queueDeclare().getQueue();
        //绑定队列到交换机
        channel.queueBind(queue,exchangeName,"info");
        channel.queueBind(queue,exchangeName,"error");
        channel.queueBind(queue,exchangeName,"warm");
        //消费消息
        channel.basicConsume(queue,new DefaultConsumer(channel){
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
                System.out.println("消费者【1】收到消息:"+new String(body));
            }
        });
        System.in.read();
    }
```

```
}
```

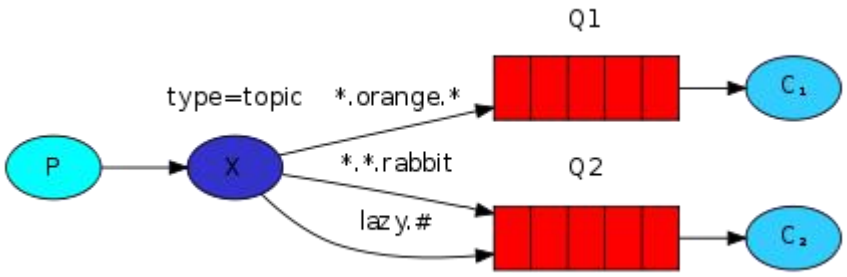
9.5. 测试

先启动两个 consumer
再使用 Producer 发送 routingKey=error 发现两个都收到
如果发送 info 发现只有 consumer2 收到
说明：它的路由 key 的匹配规则是等值匹配

10. 【掌握】第五种模型(Routing)-Topic

10.1. 概述

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



通配符

```
* (star) can substitute for exactly one word.    匹配不多不少恰好 1 个词
# (hash) can substitute for zero or more words.  匹配 0 个或多个词
```

如：

```
audit.#    匹配 audit.irs.corporate 或者 audit.irs 等
audit.*    只能匹配 audit.irs
```

10.2. 创建生产者

```
package e_routing_topic;

import com.rabbitmq.client.BuiltinExchangeType;
```

```
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import org.junit.Test;
import utils.RabbitMQUtils;

/**
 * @Author: 尚学堂 雷哥
 *
 * 交换机 topic 模式
 */
public class Producer {
    @Test
    public void sendMessage() throws   Exception{
        Connection connection = RabbitMQUtils.getConnection();
        Channel channel = connection.createChannel();
        String exchangeName = "topic";
        channel.exchangeDeclare(exchangeName, BuiltinExchangeType.TOPIC);
        //声明一个路由 key
        String routingKey="user";
        channel.basicPublish(exchangeName,routingKey,null,
                ("我是一个 topic 类型的交换机消息-routingKey:"+routingKey).getBytes());
        System.out.println("消息发送成功");
        RabbitMQUtils.closeChannelAndConnection(channel,connection);
    }
}
```

10.3. 创建消费者 1

```
package e_routing_topic;
import com.rabbitmq.client.*;
import org.junit.Test;
import utils.RabbitMQUtils;
import java.io.IOException;
/**
 * @Author: 尚学堂 雷哥
 * * 交换机直连模式
 */
public class Consumer1 {

    @Test
    public void receiveMessage() throws   Exception{
        Connection connection = RabbitMQUtils.getConnection();
```

```
        Channel channel = connection.createChannel();
        String exchangeName = "topic";
        channel.exchangeDeclare(exchangeName, BuiltinExchangeType.TOPIC);
        //得到一个临时队列
        String queue = channel.queueDeclare().getQueue();
        //绑定队列到交换机
        channel.queueBind(queue,exchangeName,"user.*");
        //消费消息
        channel.basicConsume(queue,new DefaultConsumer(channel){
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
                System.out.println("消费者【1】收到消息:"+new String(body));
            }
        });
        System.in.read();
    }
}
```

10.4. 创建消费者 2

```
package e_routing_topic;
import com.rabbitmq.client.*;
import org.junit.Test;
import utils.RabbitMQUtils;
import java.io.IOException;
/**
 * @Author: 尚学堂 雷哥
 * * 交换机直连模式
 */
public class Consumer2 {
    @Test
    public void receiveMessage() throws  Exception{
        Connection connection = RabbitMQUtils.getConnection();
        Channel channel = connection.createChannel();
        String exchangeName = "topic";
        channel.exchangeDeclare(exchangeName, BuiltinExchangeType.TOPIC);
        //得到一个临时队列
        String queue = channel.queueDeclare().getQueue();
        //绑定队列到交换机
        channel.queueBind(queue,exchangeName,"user.#");
        //消费消息
        channel.basicConsume(queue,new DefaultConsumer(channel){
```

```
                @Override
                public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
                    System.out.println("消费者【2】收到消息:"+new String(body));
                }
            });
            System.in.read();
        }
}
```

10.5. 测试

先启动消费者 1 和消费者 2
再分别发送 routingKey= user 和 user.add 和 user.add.insert 等类型的消息、
查看结果

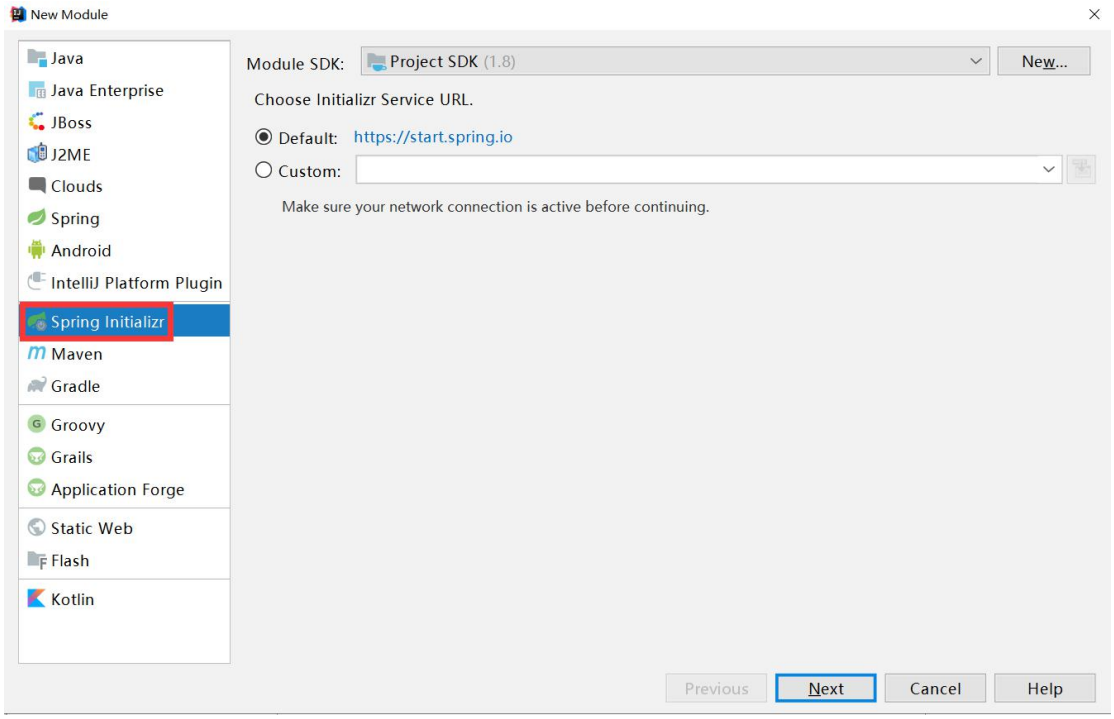
11. 【掌握】SpringBoot 中使用 RabbitMQ

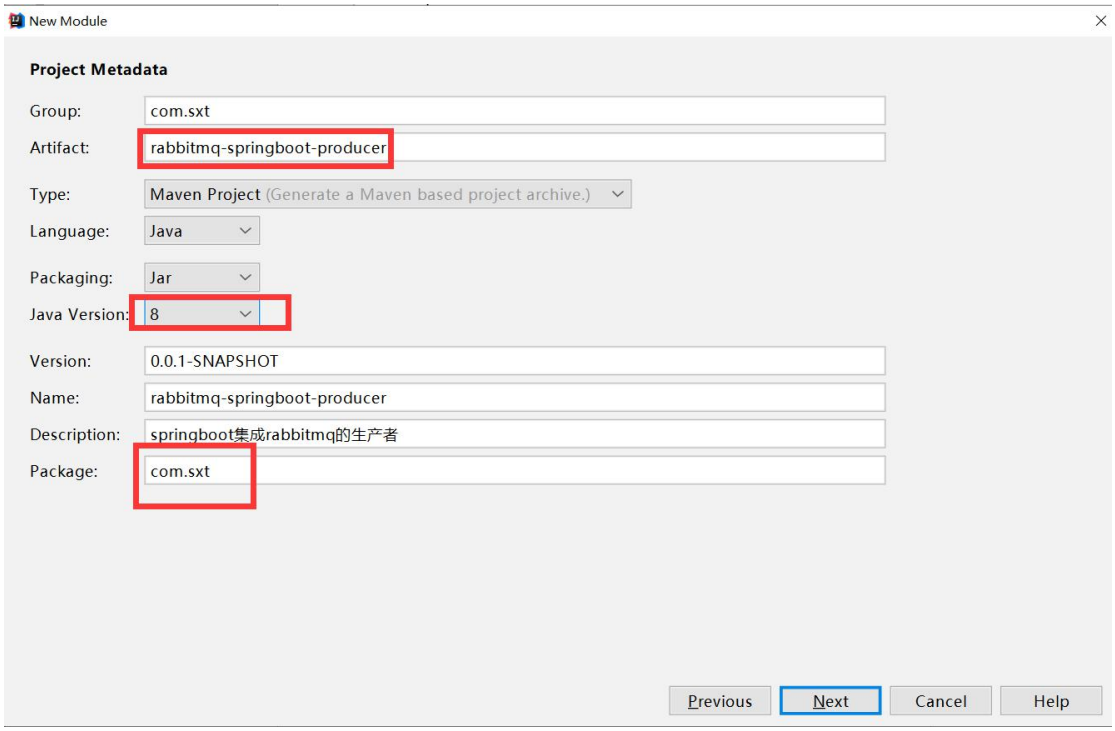
11.1. 概述

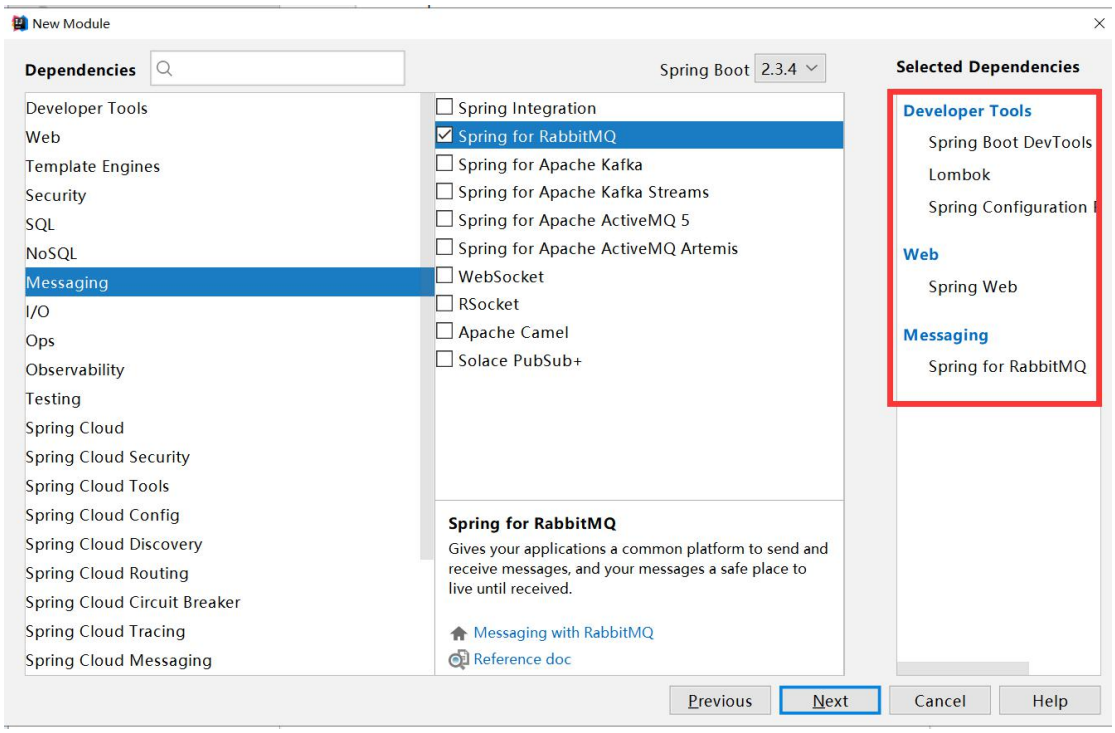
在实际开发中，我们都是使用 springboot 的技术栈来完成 RabbitMQ 的开发
这个测试案例中，我们使用两个项目，一个生产者的项目，一个消费者的项目

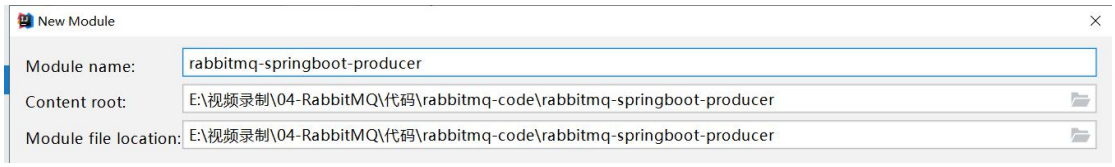
11.2. 生产者准备工作

11.2.1. 创建项目 rabbitmq-springboot-producer


New Module
Java
Java Enterprise
JBoss
J2ME
Clouds
Spring
Android
IntelliJ Platform Plugin
Spring Initializr
Maven
Gradle
Groovy
Grails
Application Forge
Static Web
Flash
Kotlin
Module SDK:
Project SDK (1.8)
New...
Choose Initializr Service URL.
Default: https://start.spring.io
Custom:
Make sure your network connection is active before continuing.
Previous
Next
Cancel
Help


New Module
Project Metadata
Group: com.sxt
Artifact: rabbitmq-springboot-producer
Type: Maven Project (Generate a Maven based project archive.)
Language: Java
Packaging: Jar
Java Version: 8
Version: 0.0.1-SNAPSHOT
Name: rabbitmq-springboot-producer
Description: springboot集成rabbitmq的生产者
Package: com.sxt
Previous
Next
Cancel
Help





11.2.2. 导入依赖

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

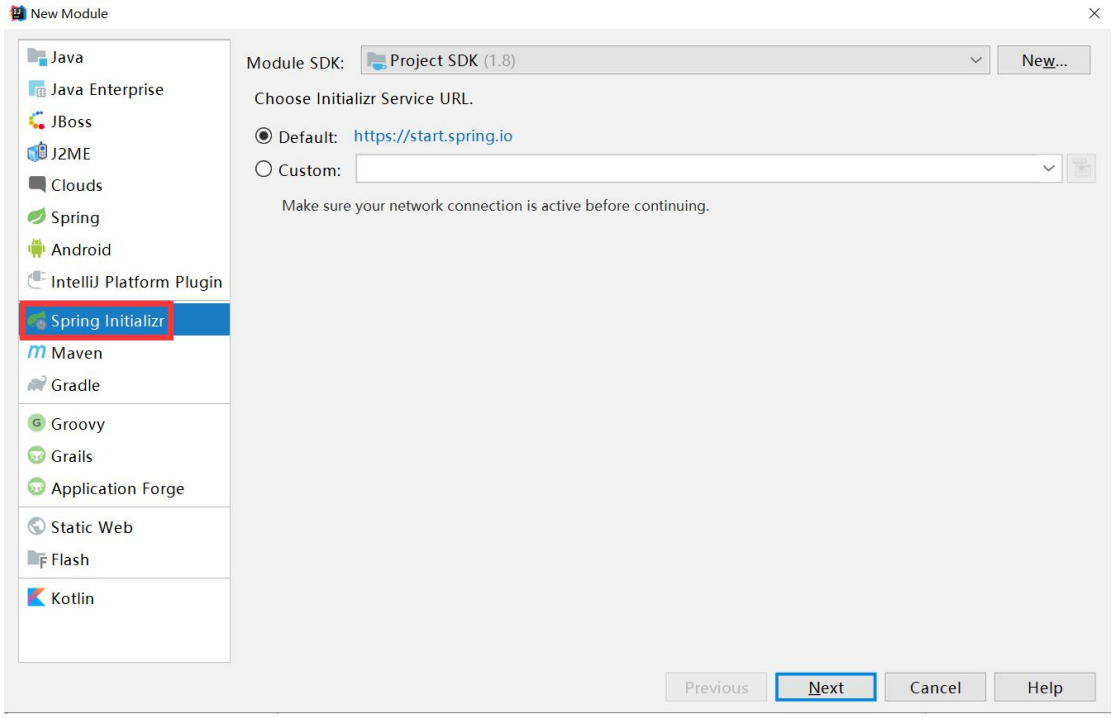
11.2.3. 修改 yml

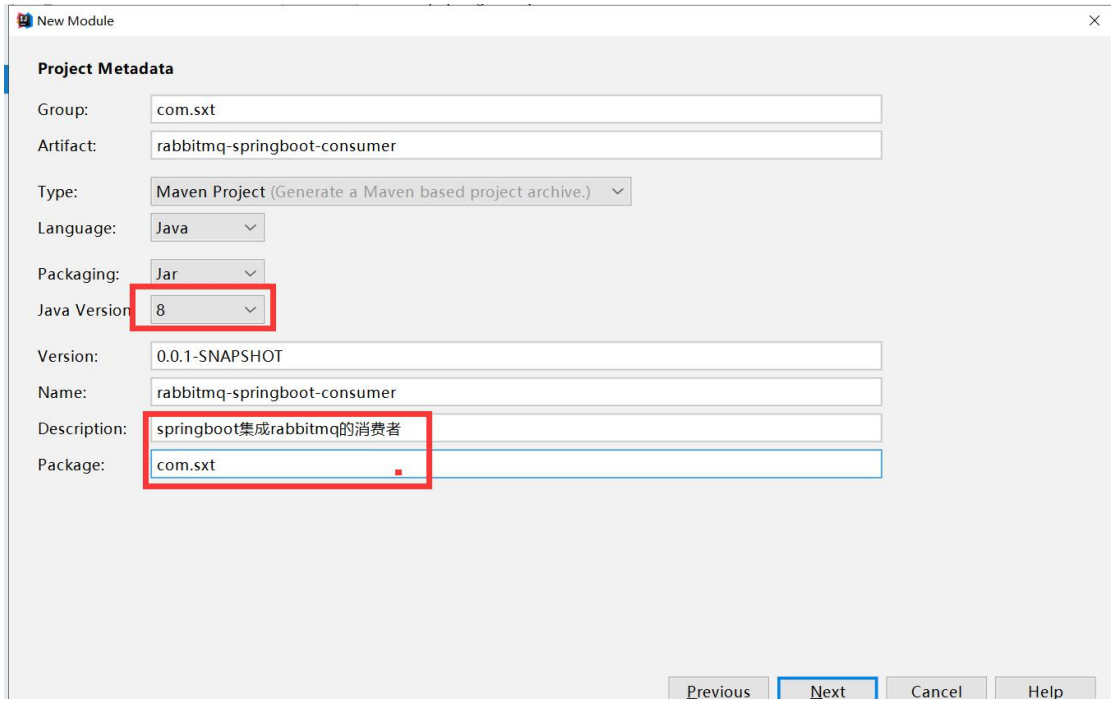
```
server:
  port: 8001
spring:
  application:
    name: producer
  rabbitmq:
    host: 116.62.44.5
    port: 5672
    username: sxt
```

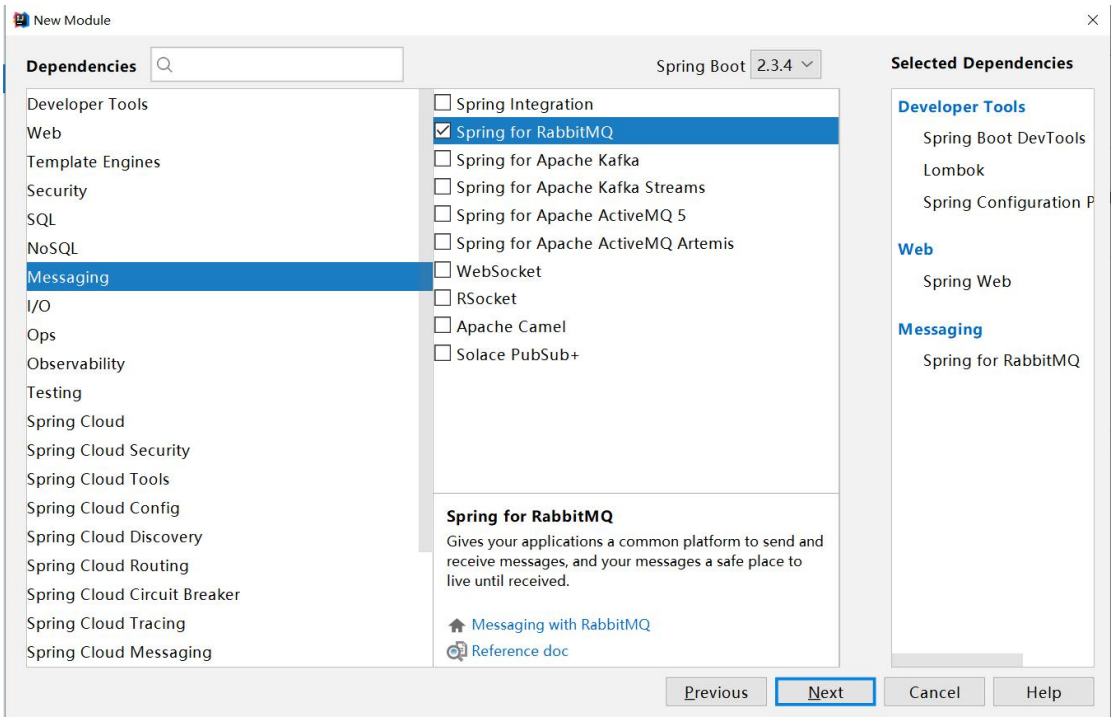
```
    password: 123456
    virtual-host: /v-sxt
```

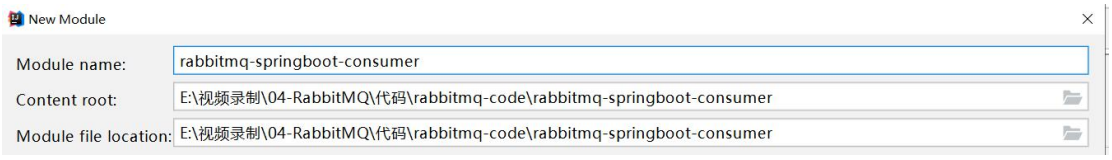
11.3. 消费者准备工作

11.3.1. 创建项目 rabbitmq-springboot-consumer









11.3.2. 导入依赖

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

11.3.3. 修改 yml

```
server:
  port: 8002
spring:
  application:
    name: producer
  rabbitmq:
    host: 116.62.44.5
    port: 5672
    username: sxt
```

```
    password: 123456
    virtual-host: /v-sxt
```

11.4. 第一种模型(直连)

11.4.1. 生产者配置

11.4.1.1. 生产者消费发送

```
/**
 * 第一种模型的消息发送
 */
@Test
void testHello(){
  rabbitTemplate.convertAndSend("hello","hello world");
  System.out.println("消息发成功");
}
```

11.4.1.2. 生产者配置类 HelloConfig

```
package com.sxt.config;

import org.springframework.amqp.core.Queue;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

/**
 * Created with IntelliJ IDEA.
 *
 * @Auther: 雷哥
 * @Date: 2020/10/07/22:21
 * @Description:
 */
@Configuration
public class HelloConfig {
  /**
   * 创建一个队列
   */
  @Bean
```

```
  public Queue hello(){
    //这里面可以和之前的Hello 项目里面一样，进行5 个参数的配置
    Queue hello = new Queue("hello");
    return hello;
  }
}
```

11.4.2. 消费者配置

11.4.2.1. 消费者接收消息两种方式都可以

```
@Component
@RabbitListener(queuesToDeclare = {@Queue("hello")})
public class HelloConsumer {

  @RabbitHandler
  public  void receive1(String message){
    System.out.println("接收到消息:"+message);
  }

}
```

```
@Component
public class HelloConsumer {

  @RabbitListener(queuesToDeclare = {@Queue("hello")})
  public  void receive1(String message){
    System.out.println("接收到消息:"+message);
  }

}
```

11.4.3. 启动测试

先使用生产者发消息，再让消费者上线，再使用生产者发消息

11.5. 第二种模型(Workquene)

11.5.1. 生产者配置

11.5.1.1. 生产者消费发送

```
/**
 * 第二模型的消息发送
 */
@Test
void  testWork(){
  for (int i = 1; i <=10 ; i++) {
    rabbitTemplate.convertAndSend("work","hello work----"+i);
  }
  System.out.println("消息发成功");
}
```

11.5.1.2. 生产者配置类 WorkConfig

```
@Configuration
public class WorkConfig {

  @Bean
  public Queue work(){
    return new Queue("work");
  }
}
```

11.5.2. 消费者配置

11.5.2.1. 消费者接收消息 WorkConsumer

```
/**
 * Created with IntelliJ IDEA.
 *
 * @Auther: 雷哥
 * @Date: 2020/10/07/1:08
 * @Description:
```

```
 */
@Component
public class WorkConsumer {

  /*
  第一个消费者
   */
  @RabbitListener(queuesToDeclare = {@Queue("work")})
  public void receive1(String message){
    System.out.println("接收到消息【1】:"+message);
  }

  /*
 *第二个消费者
  */
  @RabbitListener(queuesToDeclare = {@Queue("work")})
  public void receive2(String message){
    System.out.println("接收到消息【2】:"+message);
  }
}
```

11.5.3. 启动测试

先使用生产者发消息，再让消费者上线

11.6. 第三种模型(fanout)广播

11.6.1. 生产者配置

11.6.1.1. 生产者消费发送

```
/**
 * 第三种模型(fanout)广播
 */
@Test
public void testFanout(){
  rabbitTemplate.convertAndSend("logs","","这是日志广播");
  System.out.println("消费发送成功");
}
```

11.6.1.2. 生产者配置类 FanoutConfig

```
/**
 * Created with IntelliJ IDEA.
 *
 * @Auther: 雷哥
 * @Date: 2020/10/07/22:58
 * @Description:
 */
@Configuration
public class FanoutConfig {

  /**
   * 声明交换机
   */
  @Bean
  public FanoutExchange fanoutExchange(){
    FanoutExchange fanoutExchange=new FanoutExchange("logs");
    return fanoutExchange;
  }

  /**
   * 声明队列
   */
  @Bean
  public Queue fanoutQueue1(){
    Queue queue=new Queue("fanout_queue1");
    return queue;
  }

  /**
   * 声明队列
   */
  @Bean
  public Queue fanoutQueue2(){
    Queue queue=new Queue("fanout_queue2");
    return queue;
  }

  /**
   *把fanout_queue1 队列绑定到交换机
   */
  @Bean
```

```
  public Binding bindingQ1(){
    Binding binding= BindingBuilder.bind(fanoutQueue1())
        .to(fanoutExchange());
    return binding;
  }

  /**
   *把fanout_queue2 队列绑定到交换机
   */
  @Bean
  public Binding bindingQ2(){
    Binding binding= BindingBuilder.bind(fanoutQueue2())
        .to(fanoutExchange());
    return binding;
  }
}
```

11.6.2. 消费者配置

11.6.2.1. 消费者接收消息 FanoutCustomer

```
/**
 * Created with IntelliJ IDEA.
 *
 * @Auther: 雷哥
 * @Date: 2020/10/07/22:24
 * @Description:
 */
@Component
public class FanoutCustomer {

  @RabbitListener(bindings = @QueueBinding( //绑定队表和交换机
      value = @Queue("fanout_queue1"),//绑定队列
      exchange = @Exchange(name="logs",type = ExchangeTypes.FANOUT)//绑定交换
机和声明交换机类型
  ))
  public void receive1(String message){
    System.out.println("receive1 = " + message);
  }

  @RabbitListener(bindings = @QueueBinding(
      value = @Queue("fanout_queue2"),
```

```
      exchange = @Exchange(name="logs",type = ExchangeTypes.FANOUT)
  ))
  public void receive2(String message){
    System.out.println("receive2 = " + message);
  }
}
```

11.6.3. 启动测试

先启动消费者，再让使用生产者发消息，结果是两个方法都接到到消息

11.7. 第四种模型(routing-Direct)

11.7.1. 生产者配置

11.7.1.1. 生产者消费发送

```
/**
 * 第四种模型(routing-Direct)
 */
@Test
public void testRoutingDirect() {
  rabbitTemplate.convertAndSend("directs", "info", "info 的日志信息");
  rabbitTemplate.convertAndSend("directs", "warm", "warm 的日志信息");
  rabbitTemplate.convertAndSend("directs", "debug", "debug 的日志信息");
  rabbitTemplate.convertAndSend("directs", "error", "error 的日志信息");
  System.out.println("消息发送成功");
}
```

11.7.1.2. 生产者配置类

```
/**
 * Created with IntelliJ IDEA.
 *
 * @Auther: 雷哥
 * @Date: 2020/10/07/23:27
```

```
 * @Description:
 */
@Configuration
public class RoutingDirectConfig {

  /**
   * 声明交换机
   */
  @Bean
  public DirectExchange directExchange(){
    DirectExchange directExchange=new DirectExchange("directs");
    return directExchange;
  }

  /**
   * 声明队列1 绑定info 和warm
   */
  @Bean
  public Queue queue1(){
    return new Queue("queue1");
  }

  /**
   * 声明队列2
   */
  @Bean
  public Queue queue2(){
    return new Queue("queue2");
  }

  /**
   * 把队列1 绑定到交换机里面指定info 的路由key
   */
  @Bean
  public Binding binding11(){
    Binding binding= BindingBuilder.bind(queue1())
        .to(directExchange()).with("info");
    return binding;
  }
  /**
   * 把队列1 绑定到交换机里面指定warm 的路由key
   */
  @Bean
  public Binding binding12(){
```

```
    Binding binding= BindingBuilder.bind(queue1())
        .to(directExchange()).with("warm");
    return binding;
  }

  /**
   * 把队列2绑定到交换机里面指定debug的路由key
   */
  @Bean
  public Binding binding21(){
    Binding binding= BindingBuilder.bind(queue2())
        .to(directExchange()).with("debug");
    return binding;
  }
  /**
   * 把队列2绑定到交换机里面指定error的路由key
   */
  @Bean
  public Binding binding22(){
    Binding binding= BindingBuilder.bind(queue2())
        .to(directExchange()).with("error");
    return binding;
  }

}
```

11.7.2. 消费者配置

11.7.2.1. 消费者接收消息 RoutingDirectConsumer

```
/**
 * Created with IntelliJ IDEA.
 *
 * @Auther: 雷哥
 * @Date: 2020/10/07/23:32
 * @Description:
 */
@Component
public class RoutingDirectConsumer {

  @RabbitListener(bindings ={
```

```
    @QueueBinding(
        value = @Queue("queue1"),
        key={"info","warm"},
        exchange = @Exchange(name="directs",type = ExchangeTypes.DIRECT)
    )})
  public void receive1(String message){
    System.out.println("消费者【1】接收到消息：" + message);
  }
  @RabbitListener(bindings ={
    @QueueBinding(
        value =@Queue("queue2"),
        key={"debug","error"},
        exchange = @Exchange(name="directs",type = ExchangeTypes.DIRECT)
    )})
  public void receive2(String message){
    System.err.println("消费者【2】接收到消息：" + message);
  }

}
```

11.7.3. 启动测试

```
消费者【1】接收到消息:info的日志信息
消费者【1】接收到消息:warm的日志信息
消费者【2】接收到消息:debug的日志信息
消费者【2】接收到消息:error的日志信息
```

11.8. 第五种模型(routing-Topic)

11.8.1. 生产者配置

11.8.1.1. 生产者消费发送

```
/**
 * 第五种模型(routing-Topic)
 */
@Test
```

```
public void testTopic(){
  rabbitTemplate.convertAndSend("topics","user.save","user.save 的消息");

rabbitTemplate.convertAndSend("topics","user.save.findAll","user.save.findA
ll 的消息");
  rabbitTemplate.convertAndSend("topics","user","user 的消息");
}
```

11.8.1.2. 生产者配置类 RoutingTopicConfig

```
/**
 * Created with IntelliJ IDEA.
 *
 * @Auther: 雷哥
 * @Date: 2020/10/07/23:27
 * @Description:
 */
@Configuration
public class RoutingTopicConfig {

  /**
   * 声明交换机
   */
  @Bean
  public TopicExchange topicExchange(){
    TopicExchange topicExchange=new TopicExchange("topics");
    return topicExchange;
  }

  /**
   * 声明队列1 绑定info 和warm
   */
  @Bean
  public Queue topicQueue1(){
    return new Queue("topicQueue1");
  }

  /**
   * 声明队列2
   */
```

```
  @Bean
  public Queue topicQueue2(){
    return new Queue("topicQueue2");
  }

  /**
   * 把队列1绑定到交换机里面指定user.*的路由key
   */
  @Bean
  public Binding binding111(){
    Binding binding= BindingBuilder.bind(topicQueue1())
        .to(topicExchange()).with("user.*");
    return binding;
  }
  /**
   * 把队列2绑定到交换机里面指定user.#的路由key
   */
  @Bean
  public Binding binding222(){
    Binding binding= BindingBuilder.bind(topicQueue2())
        .to(topicExchange()).with("user.#");
    return binding;
  }

}
```

11.8.2. 消费者配置

11.8.2.1. 消费者接收消息 RoutingTopicConsumer

```
/**
 * Created with IntelliJ IDEA.
 *
 * @Auther: 雷哥
 * @Date: 2020/10/07/23:32
 * @Description:
 */
@Component
public class RoutingTopicConsumer {
```

```
  @RabbitListener(bindings ={
      @QueueBinding(
          value = @Queue("topicQueue1"),
          key={"user.*"},
          exchange = @Exchange(name="topics",type = ExchangeTypes.TOPIC)
      )})
  public void receive1(String message){
    System.out.println("消费者 user.*【1】接收到消息: " + message);
  }
  @RabbitListener(bindings ={
      @QueueBinding(
          value =@Queue("topicQueue2"),
          key={"user.#"},
          exchange = @Exchange(name="topics",type = ExchangeTypes.TOPIC)
      )})
  public void receive2(String message){
    System.err.println("消费者 user.#【2】接收到消息: " + message);
  }

}
```

11.8.3. 启动测试

12. 【掌握】boot 中发送消息的监视

12.1. 概述及准备

刚才我们发送消息，不管成功还是失败，都不报错，结果看效果时，发现有的没有发进去，那么如何知道消息是否发送成功呢，RabbitMQ 提供了一个消费监视功能

注意：RabbitMQ 发送消息分为 2 个阶段

- 消息发送到交互机里面，可以监视
- 消息由交互机到队列里面，也可以监视

12.1.1. 修改 yaml

```yaml
server:
  port: 8080
spring:
  application:
    name: springboot_rabbitmq
  rabbitmq:
    host: 116.62.44.5
    port: 5672
    username: sxt
    password: 123456
    virtual-host: /v-sxt
    publisher-confirm-type:  correlated  #开启消息到达交互机的确认机制
    publisher-returns: true  #消息由交互机到达队列时失败触发
```

12.2. 消息到交互机和未到达队列里面的监视

```java
/**
 * 第三种模型的消息发送 fanout
 */
@Test
void testFanout(){
  rabbitTemplate.setConfirmCallback(new ConfirmCallback() {
    /**
     * 当消息到达交换机之后该方法会被回调
     * @param correlationData  相关的数据
     * @param ack  交换机接收消息是否成功
     * @param cause  如果没有成功，返回原因
     */
    @Override
    public void confirm(CorrelationData correlationData, boolean ack, String cause) {
      if(ack){
        System.out.println("交换机接收消息成功");
      }else{
        System.out.println("交换机接收消息失败：原因:"+cause);
      }
    }
```

```
  });
  rabbitTemplate.setReturnCallback(new ReturnCallback() {
    /**
     * 消息如未正常到达队列里面会回调
     * @param message  消息内容
     * @param replyCode  回调的响应码
     * @param replyText  响应文本
     * @param exchange 交换机
     * @param routingKey  路由 key
     */
    @Override
    public void returnedMessage(Message message, int replyCode, String replyText, String
exchange,
        String routingKey) {
      System.out.println(message);
      System.out.println("消息未正常到达队列-replyCode:"+replyCode);
      System.out.println("消息未正常到达队列-replyText:"+replyText);
      System.out.println("消息未正常到达队列-exchange:"+exchange);
      System.out.println("消息未正常到达队列-routingKey:"+routingKey);
    }
  });
  rabbitTemplate.convertAndSend("logs","","这是一个 fanout 的日志消息");
  System.out.println("消息发送成功");
  try {
    Thread.sleep(500);
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
}
```

12.3. 测试

先去掉交换机的声明　队列的声明，队列和交换机的绑定【注释 FantoutConfig 里面的所有内容】
发消息测试

打开交换机的声明
发消息测试

打开队列的声明
发消息测试

打开队列和交换机绑定的声明

发消息测试

12.4. 优化消息监视和到达队列的监视

```
package com.sxt.config;

import org.springframework.amqp.core.Message;
import org.springframework.amqp.rabbit.connection.CorrelationData;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import javax.annotation.PostConstruct;
/**
 * @Author: 尚学堂 雷哥
 */
@Component
public class WatchMessageImpl implements
RabbitTemplate.ConfirmCallback,RabbitTemplate.ReturnCallback {

    @Autowired
    private RabbitTemplate rabbitTemplate;

    @PostConstruct
    public void initRabbitTemplate(){
        rabbitTemplate.setConfirmCallback(this);
        rabbitTemplate.setReturnCallback(this);
    }

    /**
     * 消息到达交换机的回调
     * @param correlationData
     * @param ack   是否到达交换机
     * @param cause
     */
    @Override
    public void confirm(CorrelationData correlationData, boolean ack, String cause) {
        if(ack){
            System.out.println("消息正常到达交换机");
        }else{
            System.out.println("消息没有到达交换机原因："+cause);
        }
    }
```

```
    /**
     * 消息由交换机到达队列失败之后的回调
     * @param message   消息体
     * @param replyCode 回调的状态码
     * @param replyText 文本
     * @param exchange   交换机的名称
     * @param routingKey 路由名
     */
    @Override
    public void returnedMessage(Message message, int replyCode, String replyText, String exchange, String
routingKey) {
        System.out.println(new String(message.getBody())+"到达队列失败"+replyCode+"   "+replyText+"
交换机为:"+exchange+"   路由 key:"+routingKey);
        //处理重新发送的问题
    }
}
```

12.5. 消息转换参数的问题

```
12  /**
13   * @Author: 尚学堂 雷哥
14   */
15  @Component
16  public class RabbitReceive {
17
18      @RabbitListener(bindings = {
19              @QueueBinding(value = @Queue,
20                      key = {"error"},
21                      exchange = @Exchange(name = "directs",type = ExchangeTypes.DIRECT)
22              )
23      })
24      public void receive1(User content, Message message, Channel channel){
25          System.out.println(content);
26          System.out.println(message);
27          System.out.println(channel);
28      }
29  }
30
```

1 可以传自定义对象，但是对象必须序列化
开发中一般使用json串去传对象

13. 【掌握】SpringBoot 中消息的消费及签收机制

13.1. 消息的签收机制说明

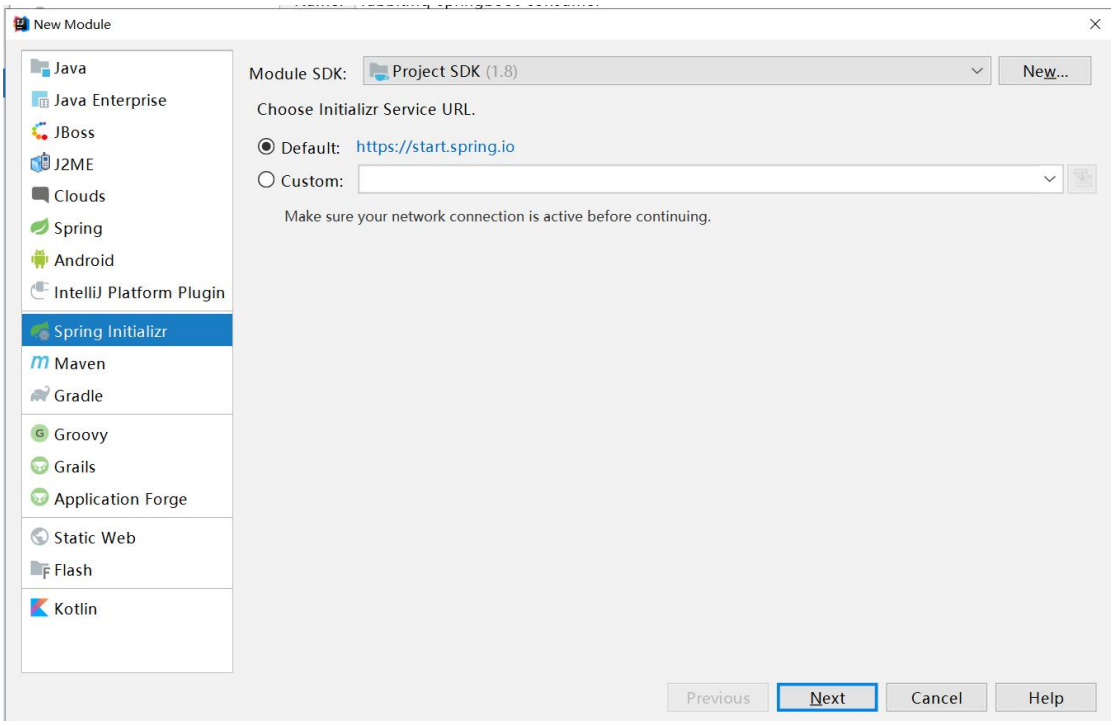
消息消费成功后，我们在客户端签收后，消息就从 MQ 服务器里面删除了
若消息没有消费成功，我们让他回到 MQ 里面，让别人再次重试消费

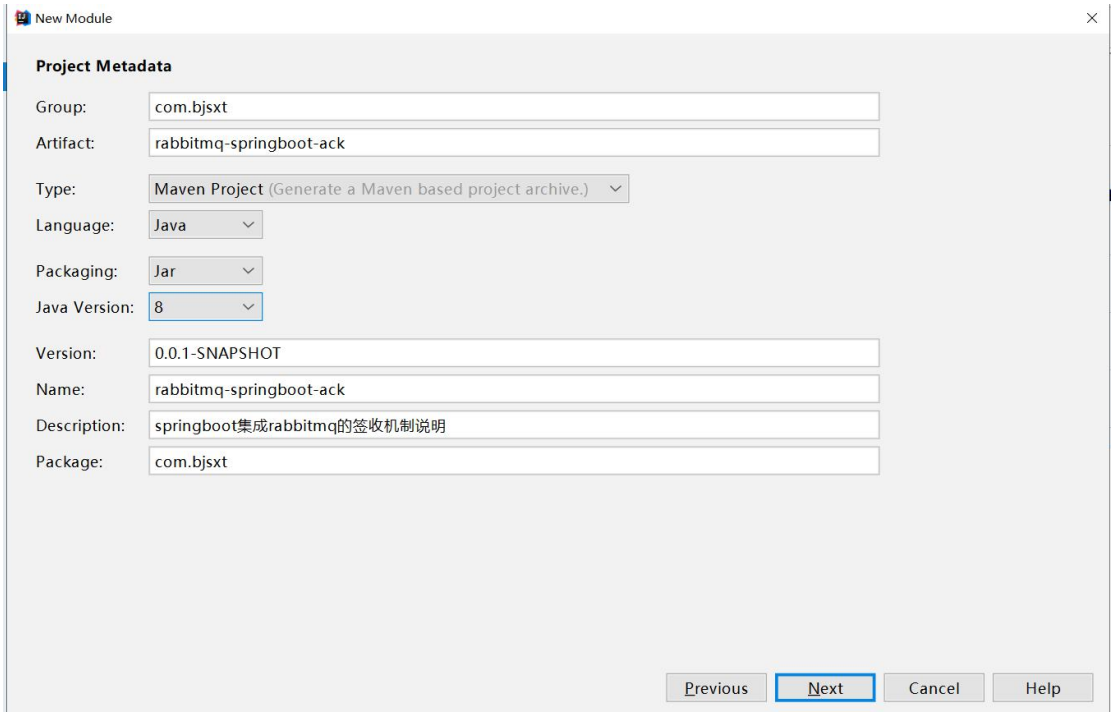
13.2. 自动签收

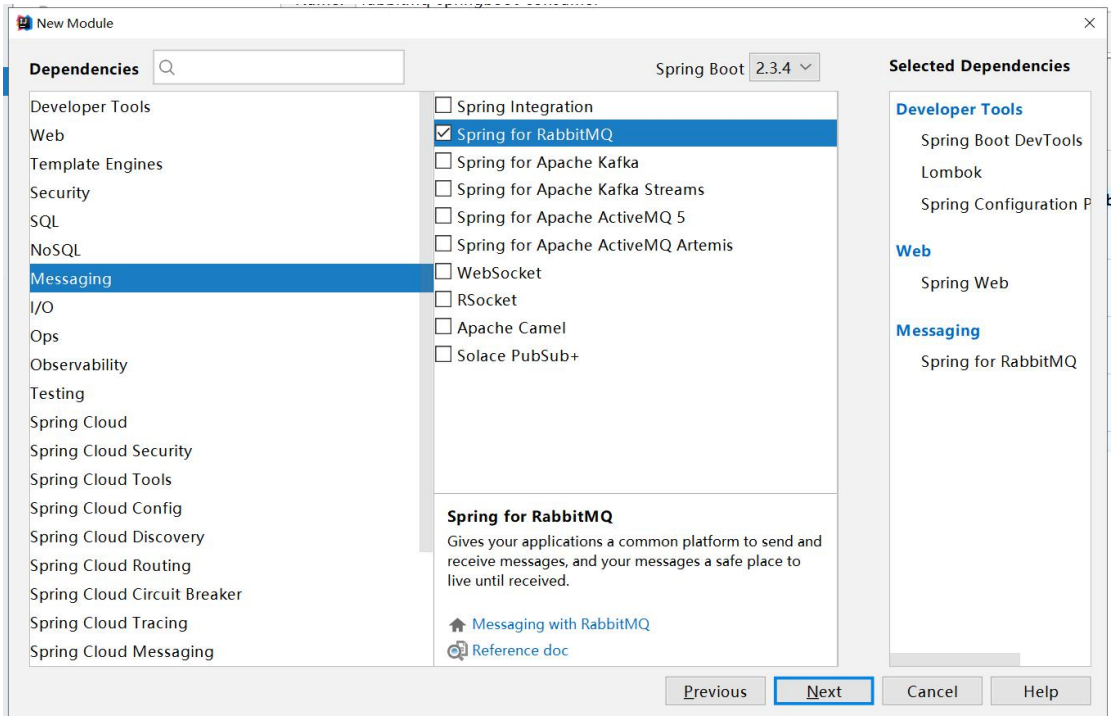
消息只要被客户端接收到，无论你客户端发生了什么，我们服务器都不管你了，直接把你删除了，这是它是默认的行为

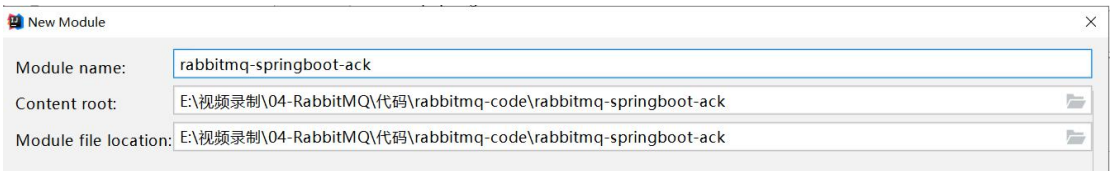
13.3. 手动签收

13.3.1. 创建项目 rabbitmq-springboot-ack





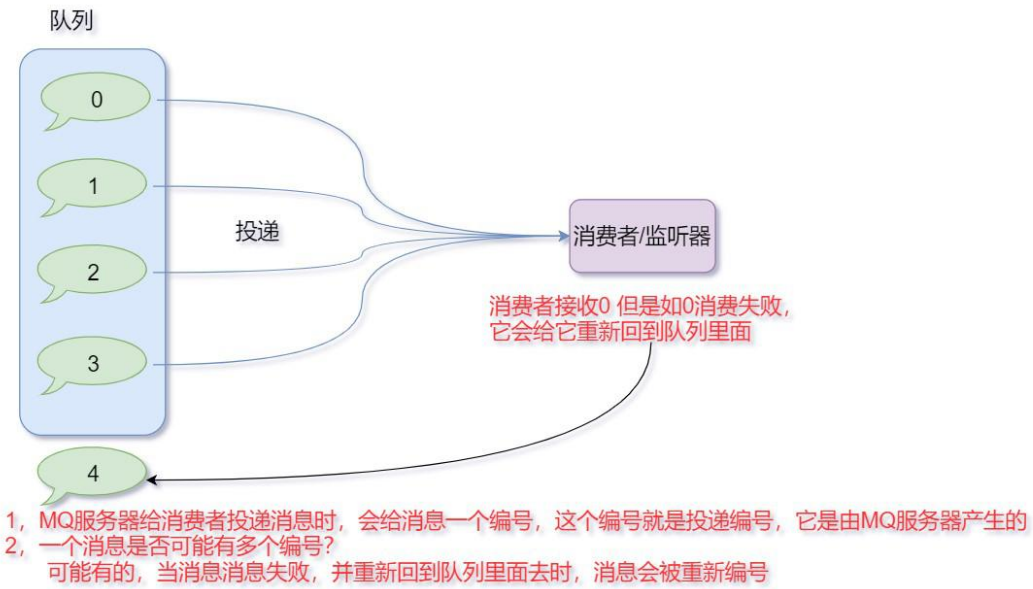




13.3.2. 创建 yml 配置文件

```yaml
server:
  port: 8001
spring:
  application:
    name: producer
  rabbitmq:
    host: 116.62.44.5
    port: 5672
    username: sxt
    password: 123456
    virtual-host: /v-sxt
    #    publisher-confirms: true   老版本里面的用法
    publisher-confirm-type: correlated  #开启消息到达交换机的确认机制
    publisher-returns: true #消息由交互机到达队列时失败触发
    listener:
      simple:
        acknowledge-mode: manual  #手动签收
      #acknowledge-mode: auto #自动签收 这个是默认行为
      direct:
        acknowledge-mode: manual #设置直连交互机的签收类型
```

13.3.3. 消息投送的 ID 的说明【重点】



13.3.4. 怎么获取投递的 ID【重点】

```python
package com.bjsxt.receive;
```

```java
import com.rabbitmq.client.Channel;
import java.io.IOException;
import org.springframework.amqp.core.Message;
import org.springframework.amqp.rabbit.annotation.Exchange;
import org.springframework.amqp.rabbit.annotation.Queue;
import org.springframework.amqp.rabbit.annotation.QueueBinding;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.stereotype.Component;

/**
 * Created with IntelliJ IDEA.
 *
 * @Auther: 雷哥
 * @Date: 2020/10/11/14:49
 * @Description:
 */
@Component
public class MessageReceive1 {
  @RabbitListener(bindings = {
      @QueueBinding(value = @Queue("queue"),
      key = "error",
      exchange = @Exchange(value = "directs"))//默认的直连交换机
  })
  public void receiveMessage(String content, Message message, Channel
channel) throws IOException {
    System.out.println("消费者收到消息-内容为:"+content);
    System.out.println("消费者收到消息-消息对象:"+message);
    System.out.println("消费者收到消息-信道:"+channel);
    long deliveryTag = message.getMessageProperties().getDeliveryTag();//
消息投递 ID
    System.out.println("消息投递 ID:"+deliveryTag);
    String messageId = message.getMessageProperties().getMessageId();
    System.out.println("消息自定义 ID:"+messageId);
    /**
     * 参数说明:
     * deliveryTag  消息投递 ID，要签收的投递 ID 是多少
     * multiple:是否批量签收
     */
    channel.basicAck(deliveryTag,false);
    System.out.println("消息签收成功");
    System.out.println("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
  }
```

```
}
```

13.3.5. 投递 ID 存在的问题及消息的永久 ID 设置的问题【重点】

什么能代表消息的唯一的标识，显然投递的 id 不行，因为一个消息可能会有多个投递的 id，我们就需要给消息一个唯一的值，这个伴随消息终身，不会变化！

我们需要发生消息时，给消息设置一个 Id，然后保证该 Id 唯一就可以！

```
  @Test
  public  void sendMessage() throws Exception {
    for (int i = 1; i <=5; i++) {
//      rabbitTemplate.convertAndSend("directs","error","我是一个日志消息
-"+i);
      rabbitTemplate.convertAndSend("directs", "error", "我是一个日志消息-" +
i,
          new MessagePostProcessor() {
            @Override
            public Message postProcessMessage(Message message) throws
AmqpException {
              //自己给消息设置自定义的ID
              String messageId= UUID.randomUUID().toString().replace("-","");
              message.getMessageProperties().setMessageId(messageId);
              return message;
            }
          });
    }
    System.out.println("消息发送成功");
    System.in.read();
  }
```

13.3.6. 关于批量的签收

若我们此时签收 4 了，但是前面 0，1，2，3 都没有签收，则 MQ 若是批量的签收，它会把 0，1，2，3 都签收，因为 MQ 认为，比他晚投递的已经签收，前面的肯定已经消费成功了

发送者

```
static int a=1;
@Test
```

```
public  void sendMessage2() throws Exception {
  for (int i = 1; i <=3; i++) {
    rabbitTemplate.convertAndSend("directs", "error", "ABC-" + i,
        new MessagePostProcessor() {
          @Override
          public Message postProcessMessage(Message message) throws AmqpException {
            //自己给消息设置自定义的ID
            message.getMessageProperties().setMessageId((a++)+"");
            return message;
          }
        });
  }
  System.out.println("消息发送成功");
  System.in.read();
}
```

消费者

```
package com.bjsxt.receive;

import com.rabbitmq.client.Channel;
import java.io.IOException;
import org.springframework.amqp.core.Message;
import org.springframework.amqp.rabbit.annotation.Exchange;
import org.springframework.amqp.rabbit.annotation.Queue;
import org.springframework.amqp.rabbit.annotation.QueueBinding;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.stereotype.Component;

/**
 * Created with IntelliJ IDEA.
 *
 * @Auther: 雷哥
 * @Date: 2020/10/11/14:49
 * @Description:
 */
@Component
public class MessageReceive2 {
  @RabbitListener(bindings = {
      @QueueBinding(value = @Queue("queue"),
      key = "error",
      exchange = @Exchange(value = "directs"))//默认的直连交换机
```

```
  })
  public void receiveMessage(String content, Message message, Channel channel) throws
IOException {
    long deliveryTag = message.getMessageProperties().getDeliveryTag();//消息投递 ID
    System.out.println("消息投递 ID:"+deliveryTag);
    String messageId = message.getMessageProperties().getMessageId();
    System.out.println("消息自定义 ID:"+messageId);
    if(content.equals("ABC-3")){
      /**
       * 参数说明:
       * deliveryTag  消息投递 ID，要签收的投递 ID 是多少
       * multiple:是否批量签收
       */
      channel.basicAck(deliveryTag,true);
      System.out.println("消息签收成功-内容为:"+content);
    }
    System.out.println("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
  }

}
```

效果

Pagination

Page 1 of 1 - Filter: Regex ?

Overview					Messages			Message rates			+/-
Virtual host	Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack	
/v-sxt	queue	classic	D	idle	0	0	0	0.00/s	0.00/s	0.00/s	

可以发现只签收了 ABC-3 但是队列里面没有消息了，说明前面的 12 都被批量签收了

13.4. 不签收

当我们认为消息不合格时，或不是我们要的消息时，我们可以选择不签收它

13.4.1. 生产者

```
@Test
public  void sendMessage2() throws Exception {
    rabbitTemplate.convertAndSend("directs", "error", "1234567",
        new MessagePostProcessor() {
          @Override
          public Message postProcessMessage(Message message) throws AmqpException {
            //自己给消息设置自定义的 ID
            String messageId= UUID.randomUUID().toString().replace("-","");
            message.getMessageProperties().setMessageId(messageId);
            return message;
          }
        });
  System.out.println("消息发送成功");
  System.in.read();
}
```

13.4.2. 消费者

```
package com.bjsxt.receive;

import com.rabbitmq.client.Channel;
import java.io.IOException;
import org.springframework.amqp.core.Message;
import org.springframework.amqp.rabbit.annotation.Exchange;
```

```java
import org.springframework.amqp.rabbit.annotation.Queue;
import org.springframework.amqp.rabbit.annotation.QueueBinding;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.stereotype.Component;

/**
 * Created with IntelliJ IDEA.
 *
 * @Auther: 雷哥
 * @Date: 2020/10/11/14:49
 * @Description:
 */
@Component
public class MessageReceive3 {
  @RabbitListener(bindings = {
      @QueueBinding(value = @Queue("queue"),
      key = "error",
      exchange = @Exchange(value = "directs"))//默认的直连交换机
  })
  public void receiveMessage(String content, Message message, Channel channel) throws
IOException {
    long deliveryTag = message.getMessageProperties().getDeliveryTag();//消息投递 ID
    System.out.println("消息投递 ID:"+deliveryTag);
    String messageId = message.getMessageProperties().getMessageId();
    System.out.println("消息自定义 ID:"+messageId);
    if(content.equals("123456")){//如果消息内容为 123456 就签收它
      /**
       * 参数说明:
       * deliveryTag  消息投递 ID，要签收的投递 ID 是多少
       * multiple:是否批量签收
       */
      channel.basicAck(deliveryTag,false);
      System.out.println("消息签收成功");
    }else{
      //如果不是 123456 就拒绝签收
      /**
       * 参数说明:
       * deliveryTag  消息投递 ID，要签收的投递 ID 是多少
       * multiple:是否批量签收
       * requeue: true 代表拒绝签收并把消息重新放回到队列里面  false 就直接拒绝
       */
      channel.basicNack(deliveryTag,false,true);
      System.out.println("消息被拒绝签收");
    }
```

```
    System.out.println("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
  }
}
```

我们选择不签收，其实是为了保护消息，当消费消息发送异常时，我们可以把消息放在队列里面，让它重新投递，重新让别人消费！而不是丢了它！

13.5. 不去签收消息的死循环怎么解决

不签收，并且让它回到队列里面，想法很好，但是很容易造成死循环，因为没有任何人能消费她！
我们设计一个机制，当一个消息被消费 3 次还没有消费成功，我们就直接把它记录下来，人工处理！
消息消费 3 次（消息的标识，消息的计数）
我们引入 Redis ，使用 Redis 计数，若超过 3 次，直接拒绝消息，并且不回到队列里面

13.5.1. 引入 redis 并使用 docker 运行

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

```
docker run -d --name myredis -p 6390:6379 redis --requirepass "123456"
```

13.5.2. 配置文件

```
#redis 的配置
redis:
  host: 116.62.44.5
  port: 6390
  password: 123456
```

13.5.3. 代码

```
package com.bjsxt.receive;

import com.rabbitmq.client.Channel;
import java.io.IOException;
import org.springframework.amqp.core.Message;
import org.springframework.amqp.rabbit.annotation.Exchange;
```

```java
import org.springframework.amqp.rabbit.annotation.Queue;
import org.springframework.amqp.rabbit.annotation.QueueBinding;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.redis.core.StringRedisTemplate;
import org.springframework.stereotype.Component;

/**
 * Created with IntelliJ IDEA.
 *
 * @Auther: 雷哥
 * @Date: 2020/10/11/14:49
 * @Description:
 */
@Component
public class MessageReceive3 {

  @Autowired
  private StringRedisTemplate redisTemplate;

  //消息的前缀
  private String MESSAGE="message:";

  @RabbitListener(bindings = {
      @QueueBinding(value = @Queue("queue"),
      key = "error",
      exchange = @Exchange(value = "directs"))//默认的直连交换机
  })
  public void receiveMessage(String content, Message message, Channel channel)
throws IOException {
    long deliveryTag = message.getMessageProperties().getDeliveryTag();//消息
投递 ID
    System.out.println("消息投递 ID:"+deliveryTag);
    String messageId = message.getMessageProperties().getMessageId();
    System.out.println("消息自定义 ID:"+messageId);
    if(content.equals("123456")){//如果消息内容为 123456 就签收它
      /**
       * 参数说明:
       * deliveryTag  消息投递 ID，要签收的投递 ID 是多少
       * multiple:是否批量签收
       */
      channel.basicAck(deliveryTag,false);
      System.out.println("消息签收成功");
    }else{
```

```
        String count=redisTemplate.opsForValue().get(MESSAGE+messageId);
        if(count!=null&&Long.valueOf(count)>=3){
          channel.basicNack(deliveryTag,false,false);
          System.out.println("该消息消费 3 失败，我们记录它，人工处理:"+content);
        }else {
          //如果不是 123456 就拒绝签收
          /**
           * 参数说明:
           * deliveryTag  消息投递 ID，要签收的投递 ID 是多少
           * multiple:是否批量签收
           * requeue: true 代表拒绝签收并把消息重新放回到队列里面  false 就直接拒绝
           */
          //处理业务逻辑的处理[可能逻辑出现问题]
          channel.basicNack(deliveryTag, false, true);
          System.out.println("消息被拒绝签收");
          //因现被拒绝了，我们把消息 ID 放到 redis 里面
          redisTemplate.opsForValue().increment(MESSAGE + messageId);
        }
      }
      System.out.println("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
    }

}
```

13.5.4. 测试注意

因为统计计数时，消息的次数，是通过消息的 Id 来计数的，我们在发送消息时，要设置消息的头：

```
@Test
public  void sendMessage2() throws Exception {
    rabbitTemplate.convertAndSend( exchange: "directs",  routingKey: "error",  message: "1234567",
        new MessagePostProcessor() {
          @Override
          public Message postProcessMessage(Message message) throws AmqpException {
            //自己给消息设置自定义的ID
            String messageId= UUID.randomUUID().toString().replace( target: "-", replacement: "");
            message.getMessageProperties().setMessageId(messageId);
            return message;
          }
        });
  System.out.println("消息发送成功");
  System.in.read();
}
```

14. 【掌握】消息的重复消费及处理【面试】

14.1. 重复消费概述

当消息回退到队列里面后，会被再次消费，但是我们不能让消息消费成功 2 次
其实,MQ 自己就可以保证消息不被重复消费，因为 MQ 可以把消息投递给消费者时，是阻塞的，不会把一个消息投递给多个消费者！
但是面试时，有人问你，消息怎么保证不被重复消费！
无论在 RabbitMQ ,或者 Activemq 里面，解决思路都是一样的！！
对于重复消费--> 去重操作--->
接口的幂等操作（->找到该操作的有个唯一标识-> 具体的情况，具体讨论）
Eg：微信里面，若有人关注我了，我给他发红包，我的红包只能发送一次，我们可以使用用户的 openId 做一个去重的操作
订单不重复（订单的编号）（先按订单编号查询，再操作）
数据库不能重复（数据库的 id）（先查询是否存在，再进行操作）
总结：去重操作就是找到一个该操作的唯一标识，把该标识，放在一个空间里面，在此操作时，我们可以先判断该空间是否已经操作过它了

14.2. 利用 BloomFilter 实现去重的操作

https://www.hutool.cn/docs/#/bloomFilter/%E6%A6%82%E8%BF%B0

14.2.1. 添加 Hutool 的依赖

```
<dependency>
  <groupId>cn.hutool</groupId>
  <artifactId>hutool-all</artifactId>
  <version>5.1.5</version>
</dependency>
```

14.2.2. 配置 BitMapBloomFilter

```
package com.bjsxt.config;
```

```
import cn.hutool.bloomfilter.BitMapBloomFilter;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

/**
 * Created with IntelliJ IDEA.
 *
 * @Auther: 雷哥
 * @Date: 2020/10/11/16:34
 * @Description:
 */
@Configuration
public class BloomFilterConfig {

  @Bean
  public BitMapBloomFilter bitMapBloomFilter(){
    return new BitMapBloomFilter(Integer.MAX_VALUE);
  }

}
```

14.2.3. 使用 Hutool -BloomFilter 去重

```
package com.bjsxt.receive;

import cn.hutool.bloomfilter.BitMapBloomFilter;
import com.rabbitmq.client.Channel;
import java.io.IOException;
import org.springframework.amqp.core.Message;
import org.springframework.amqp.rabbit.annotation.Exchange;
import org.springframework.amqp.rabbit.annotation.Queue;
import org.springframework.amqp.rabbit.annotation.QueueBinding;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.redis.core.StringRedisTemplate;
import org.springframework.stereotype.Component;

/**
 * Created with IntelliJ IDEA.
 *
 * @Auther: 雷哥
```

```
 * @Date: 2020/10/11/14:49
 * @Description:
 */
@Component
public class MessageReceive4 {

  @Autowired
  private StringRedisTemplate redisTemplate;

  @Autowired
  private BitMapBloomFilter bitMapBloomFilter;

  //消息的前缀
  private String MESSAGE="message:";

  @RabbitListener(bindings = {
      @QueueBinding(value = @Queue("queue"),
      key = "error",
      exchange = @Exchange(value = "directs"))//默认的直连交换机
  })
  public void receiveMessage(String content, Message message, Channel channel)
throws IOException {
    long deliveryTag = message.getMessageProperties().getDeliveryTag();//消息
投递 ID
    System.out.println("消息投递 ID:"+deliveryTag);
    String messageId = message.getMessageProperties().getMessageId();
    System.out.println("消息自定义 ID:"+messageId);

    if(bitMapBloomFilter.contains(messageId)){
      System.out.println("这个消息被消费过，不能重复消费");
      try {
        //如果进入到这里面，说明这个消息之前被消费过，但是 MQ 认为你没有消费，所以我
们要签收这条消息
        channel.basicAck(deliveryTag,false);
        return;
      }catch (Exception e){
        System.out.println(e);
      }
    }

    if(content.equals("123456")){//如果消息内容为 123456 就签收它
      /**
       * 参数说明:
       * deliveryTag  消息投递 ID，要签收的投递 ID 是多少
```

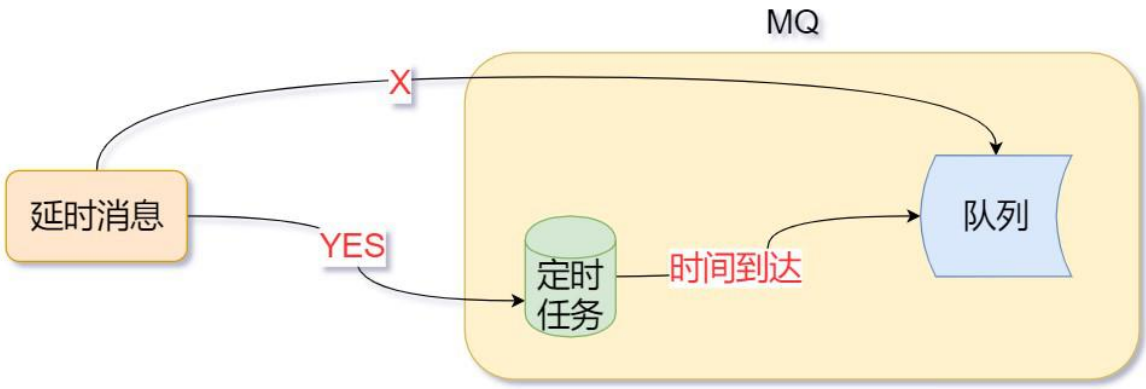
```
        * multiple:是否批量签收
        */
      channel.basicAck(deliveryTag,false);
      System.out.println("消息签收成功");
      //消费成功之后放到布隆过滤器里面
      bitMapBloomFilter.add(messageId);
    }else{
      String count=redisTemplate.opsForValue().get(MESSAGE+messageId);
      if(count!=null&&Long.valueOf(count)>=3){
        channel.basicNack(deliveryTag,false,false);
        System.out.println("该消息消费 3 失败，我们记录它，人工处理:"+content);
      }else {
        //如果不是 123456 就拒绝签收
        /**
         * 参数说明:
         * deliveryTag  消息投递 ID，要签收的投递 ID 是多少
         * multiple:是否批量签收
         * requeue: true 代表拒绝签收并把消息重新放回到队列里面  false 就直接拒绝
         */
        //处理业务逻辑的处理[可能逻辑出现问题]
        channel.basicNack(deliveryTag, false, true);
        System.out.println("消息被拒绝签收");
        //因现被拒绝了，我们把消息 ID 放到 redis 里面
        redisTemplate.opsForValue().increment(MESSAGE + messageId);
      }
    }
    System.out.println("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
  }

}
```

14.2.4. 发消息的代码和上面一样

15. 【掌握】延迟消息+死信消息（常考）

15.1. 什么是延迟消息及使用场景

当消息发送到服务器时，该消息不能直接被放在队列里面，而是在 mq 服务器里面建立一个定时任务，在服务器到达延时时间后再执行投递的操作



15.1.1. 延迟消息的使用场景

淘宝七天自动确认收货。在我们签收商品后，物流系统会在七天后延时发送一个消息给支付系统，通知支付系统将款打给商家，这个过程持续七天，就是使用了消息中间件的延迟推送功能。

12306 购票支付确认页面。我们在选好票点击确定跳转的页面中往往都会有倒计时，代表着 30 分钟内订单不确认的话将会自动取消订单。其实在下订单那一刻开始购票业务系统就会发送一个延时消息给订单系统，延时 30 分钟，告诉订单系统订单未完成，如果我们在 30 分钟内完成了订单，则可以通过逻辑代码判断来忽略掉收到的消息。

在上面两种场景中，如果我们使用下面两种传统解决

1 对于单机版而已，我们的策略有 3 种

定时删除、定期删除、惰性删除

2 对于分布式项目而言呢，我们最好选择 MQ

插件安装：https://www.cnblogs.com/geekdc/p/13549613.html

15.2. 对于 Activemq 实现延迟消息【可以回看雷哥的 ActiveMQ】

15.2.1. 修改配置文件

```
    </bean>

    <!--
        The <broker> element is used to configure the ActiveMQ broker.
    -->
    <broker xmlns="http://activemq.apache.org/schema/core" schedulerSupport="true" brokerName="localhost" dataDirectory="${acti
vemq.data}">

        <destinationPolicy>
            <policyMap>
              <policyEntries>
                <policyEntry topic=">" >
                    <!-- The constantPendingMessageLimitStrategy is used to prevent
```

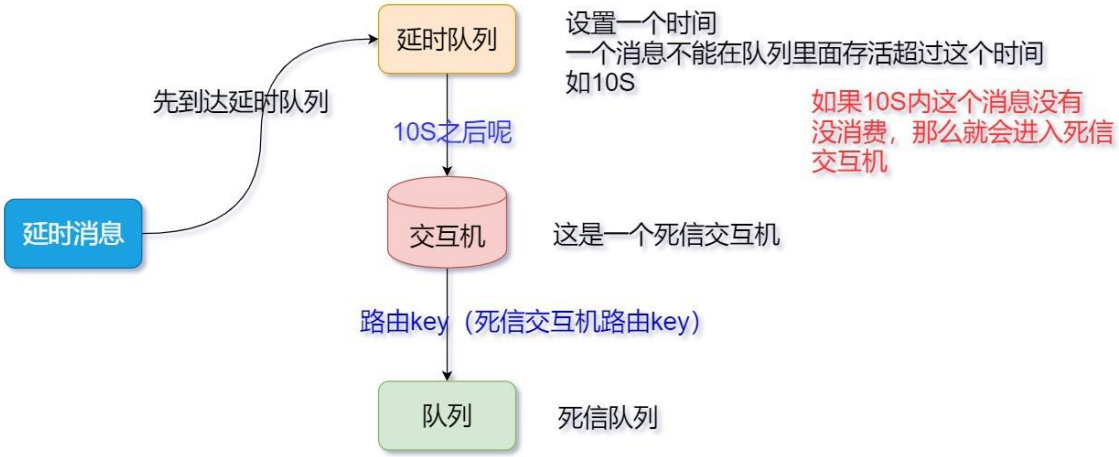
15.2.2. 发消息时，指定时间

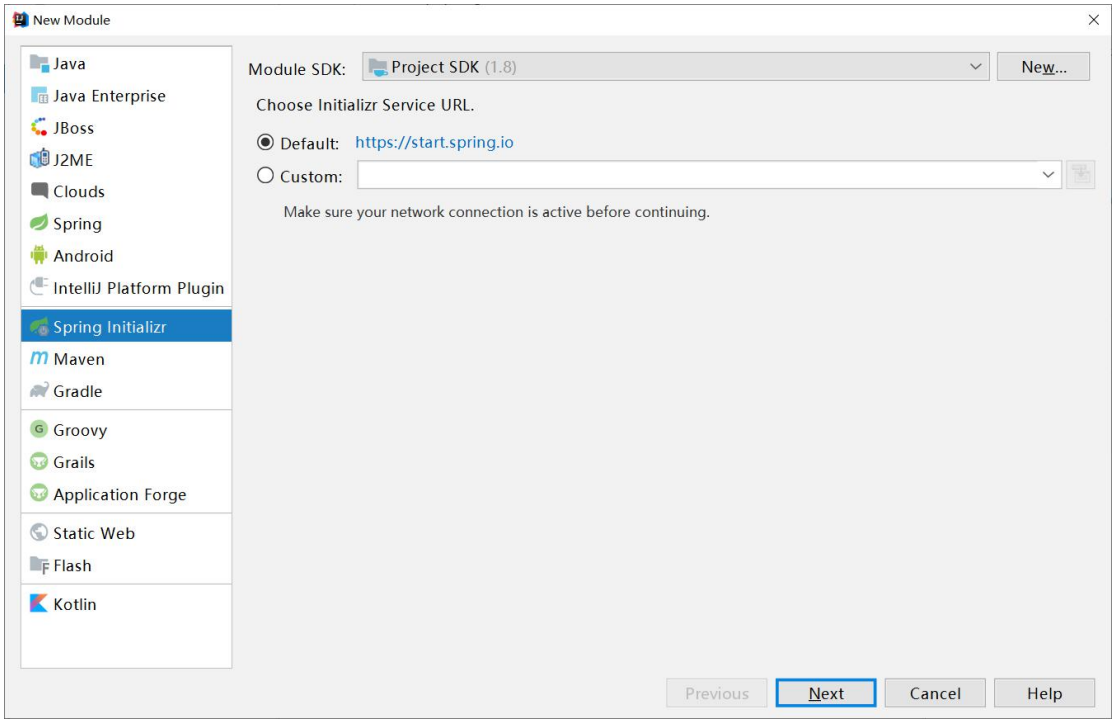
```
MessageProducer producer = session.createProducer(destination);
TextMessage message = session.createTextMessage("test msg");
long time = 60 * 1000;
message.setLongProperty(ScheduledMessage.AMQ_SCHEDULED_DELAY, time);
producer.send(message);
```

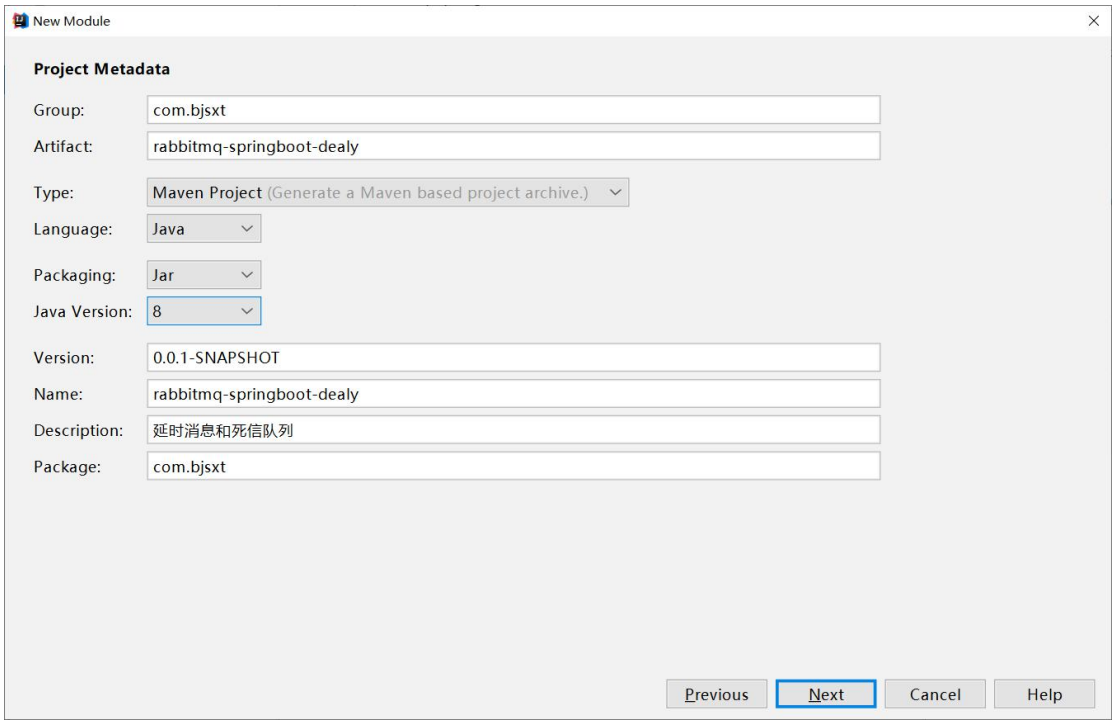
15.3. Rabbitmq 的延时消息实现

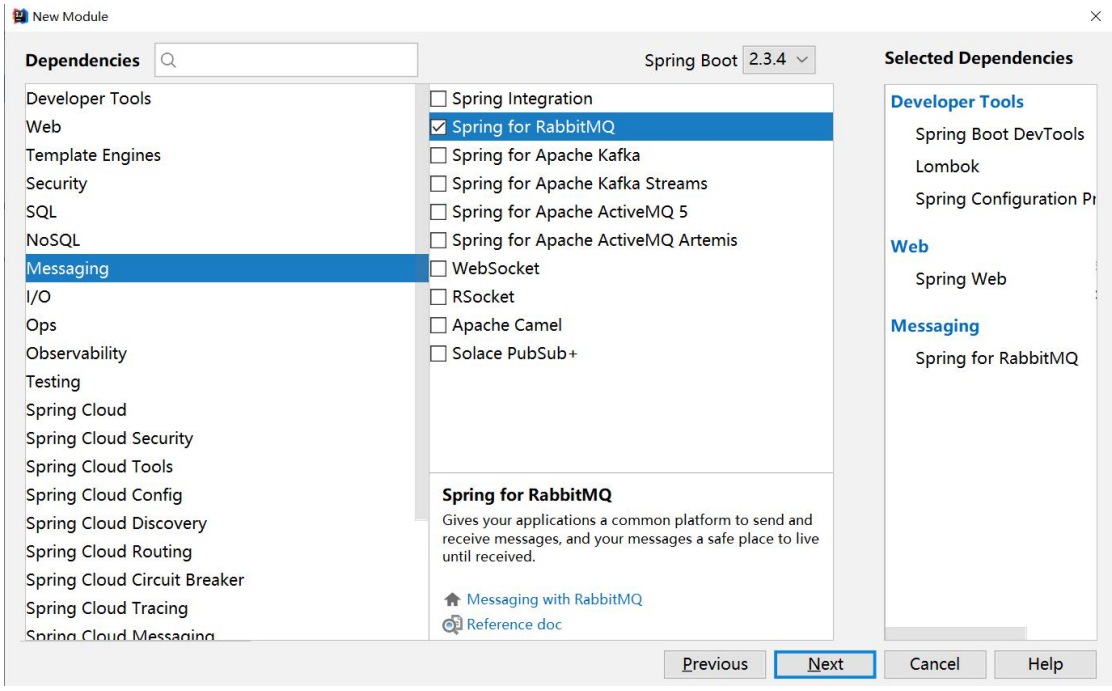
Rabbitmq 本身没有提供一个队列的机制，但是它里面有个延迟队列的机制

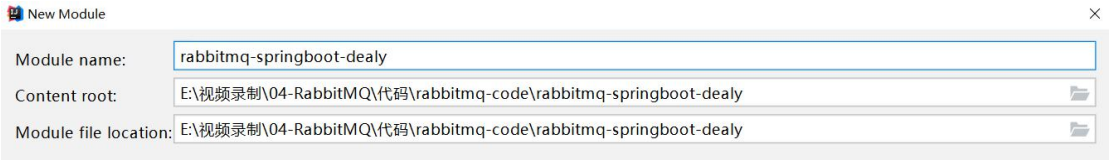
15.3.1. 延迟队列的机制及创建项目 rabbitmq-springboot-dealy




New Module
Java
Java Enterprise
JBoss
J2ME
Clouds
Spring
Android
IntelliJ Platform Plugin
Spring Initializr
Maven
Gradle
Groovy
Grails
Application Forge
Static Web
Flash
Kotlin
Module SDK:
Project SDK (1.8)
New...
Choose Initializr Service URL.
Default: https://start.spring.io
Custom:
Make sure your network connection is active before continuing.
Previous
Next
Cancel
Help


New Module
Project Metadata
Group: com.bjsxt
Artifact: rabbitmq-springboot-dealy
Type: Maven Project (Generate a Maven based project archive.)
Language: Java
Packaging: Jar
Java Version: 8
Version: 0.0.1-SNAPSHOT
Name: rabbitmq-springboot-dealy
Description: 延时消息和死信队列
Package: com.bjsxt
Previous
Next
Cancel
Help





15.3.2. 创建 yml 文件

```yml
server:
  port: 8001
spring:
  application:
    name: rabbitmq-springboot-dealy
  rabbitmq:
    host: 116.62.44.5
    port: 5672
    username: sxt
    password: 123456
    virtual-host: /v-sxt
    #    publisher-confirms: true  老版本里面的用法
    publisher-confirm-type: correlated  #开启消息到达交换机的确认机制
    publisher-returns: true #消息由交互机到达队列时失败触发
    listener:
      simple:
        acknowledge-mode: manual  #手动签收
      #acknowledge-mode: auto #自动签收 这个是默认行为
      direct:
```

```
        acknowledge-mode: manual #设置直连交互机的签收类型
```

15.3.3. 代码实现延迟消息

```
package com.bjsxt.config;

import java.util.HashMap;
import org.springframework.amqp.core.Binding;
import org.springframework.amqp.core.BindingBuilder;
import org.springframework.amqp.core.DirectExchange;
import org.springframework.amqp.core.Queue;
import org.springframework.context.annotation.Bean;

/**
 * Created with IntelliJ IDEA.
 *
 * @Auther: 雷哥
 * @Date: 2020/10/11/16:43
 * @Description:
 */
public class DealyMessageConfig {
  @Bean
  public Queue dealyQueue(){
    HashMap<String, Object> args = new HashMap<>();
    // 把一个队列修改为延迟队列
    args.put("x-message-ttl",10*1000) ; // 消息的最大存活时间
    args.put("x-dead-letter-exchange","DeadLetter.exc") ; // 该队列里面的消息死了，去那个交换机
    args.put("x-dead-letter-routing-key","DeadLetter.key") ; // 该队列里面的消息死了，去那个交换机，由那个路由key 路由他
    Queue dealy = new Queue("dealy",true,false,false,args);
    return dealy ;
  }

  /**
   * 死信交互就
   * @return
   */
  @Bean
  public DirectExchange deadLetterExchange() {
    return new DirectExchange("DeadLetter.exc");
```

```
    }


    /**
     * 绑定
     * @return
     */
    @Bean
    public Binding newAndDeadLetterExchange(){
      return BindingBuilder.bind(newQueue()).to(deadLetterExchange()).
          with("DeadLetter.key"); // 死信路由key
    }

    /**
     * 新生队列
     * @return
     */
    @Bean
    public Queue newQueue(){
      return  new Queue("new.queue") ;
    }
}
```

15.3.4. 监听新生的世界

```
package com.bjsxt.receive;

import com.rabbitmq.client.Channel;
import java.io.IOException;
import java.text.SimpleDateFormat;
import java.util.Date;
import org.springframework.amqp.core.Message;
import org.springframework.amqp.rabbit.annotation.RabbitHandler;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.stereotype.Component;

/**
 * Created with IntelliJ IDEA.
 *
 * @Auther: 雷哥
 * @Date: 2020/10/11/14:49
```

```
 * @Description:
 */
@Component
@RabbitListener(queues = {"new.queue"})
public class MessageReceive5 {
  @RabbitHandler
  public void onMessage(String content, Message message, Channel channel)
{
    System.out.println("来到了新的世界，正在消费中。。。");
    long deliveryTag = message.getMessageProperties().getDeliveryTag();

    try {
      channel.basicAck(deliveryTag, false);
      System.out.println("签收成功：" + content);
      SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
      System.out.println("签收成功时间:" + sdf.format(new Date()));
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

15.3.5. 测试

我们往延迟队列发送消息

启动项目

```
@Test
public void sendDealy() throws IOException {
  SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
  System.out.println("消息发送时间:"+sdf.format(new Date()));
  rabbitTemplate.convertAndSend("dealy","我是一个延时消息");
  System.out.println("消息发送成功");
  System.in.read();
}
```

```
消息发送时间:2020-09-19 02:12:54
消息发送成功
来到了新的世界，正在消费中。。。
签收成功：我是一个延时消息
签收成功时间:2020-09-19 02:13:04
```

15.3.6. 发消息时，也可以指定时间

我们的这个队列里面让你活 10s ，你想活 11s 不行
但是你活 5s 是可以的
它是以那个最小为主的！

```
@Test
public void sendDealy2() throws IOException {
  SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
  System.out.println("消息发送时间:"+sdf.format(new Date()));
  rabbitTemplate.convertAndSend("dealy", (Object) "我是一个延时消息", new
MessagePostProcessor() {
    @Override
    public Message postProcessMessage(Message message) throws AmqpException {
      message.getMessageProperties().setExpiration("5000");//我们延时队列里面
设置的为10 秒，但这个消息它只想活 5 秒
      return message;
    }
  });
  System.out.println("消息发送成功");
  System.in.read();
}
```

All queues (2)

Pagination

Page 1 of 1 - Filter: Regex ?

Overview					Messages			Message rates			+/-
Virtual host	Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack	
/v-sxt	dealy	classic	D TTL DLX DLK	idle	0	0	0	0.00/s			
/v-sxt	new.queue	classic	D	idle	0	0	0		0.00/s	0.00/s	

Add a new queue