# 04-rabbitmq-工作队列-spring

## 先决条件

本教程假定RabbitMQ已在标准端口（5672）上的localhost上安装并运行。如果使用不同的主机，端口或凭据，连接设置将需要调整。

## 工作队列（使用spring-ampq客户端）



在第一个教程中，我们编写了程序来发送和接收来自命名队列的消息。在这一个中，我们将创建一个工作队列，用于在多个工作人员之间分配耗时的任务。

工作队列（又称：任务队列）背后的主要思想是避免立即执行资源密集型，并且必须等待完成的任务。相反，我们安排任务在后续完成。我们将任务封装成 消息，并将其发送到队列。在后台运行的工作进程将弹出任务并最终执行作业。当你运行很多工作进程时，这些任务将在它们之间共享。

这个概念在Web应用程序中特别有用，尤其适用在短时间HTTP请求窗口中无法处理复杂的任务。

## 准备

在本教程的前面部分，我们发送了一个包含“Hello World!”的消息。现在我们将发送的的字符串代表复杂任务。我们没有一个现实世界的任务，比如图像被调整大小，或者是要渲染的pdf文件来模拟，所以假设我们很忙 - 通过使用Thread.sleep()函数来假冒它。我们将把字符串中的点数作为其复杂度; 每个点都将占“工作”的一秒钟。例如，由Hello ...描述的假任务将需要三秒钟。

如果您尚未设置项目，请参阅第一个教程中的设置。我们将遵循与第一个教程中相同的模式：1）创建一个包（com.example.rabbitmq）并创建一个Tut2Config，Tut2Receiver和Tut2Sender。首先创建一个新包（com.example.rabbitmq），我们将放置我们的三个类。在配置类中，我们设置两个配置文件，教程的标签（“tut2”）和模式的名称（“work-queues”）。我们利用spring来将队列暴露为一个bean。我们将接收者设置为配置文件，并定义两个bean，以对应于上图中的工作进程; receiver1和receiver2。最后，我们定义发件者的配置文件并定义发件者bean。配置现在完成了。

|  |
| --- |
| **package** com.example.rabbitmq;  **import** org.springframework.amqp.core.Queue; **import** org.springframework.context.annotation.Bean; **import** org.springframework.context.annotation.Configuration; **import** org.springframework.context.annotation.Profile;  */\*\*  \* RabbitMQ消息配置  \* Author: 王俊超  \* Date: 2017-06-17 10:26  \* All Rights Reserved !!!  \*/* @Profile({**"tut2"**, **"work-queues"**}) @Configuration **public class** Tut2Config {  */\*\*  \* 创建一个消息队列  \** ***@return*** *\*/* @Bean  **public** Queue hello() {  **return new** Queue(**"hello"**);  }   */\*\*  \* 定一个接收配置对象，定义两个接收者  \*/* @Profile(**"receiver"**)  **private static class** ReceiverConfig {   @Bean  **public** Tut2Receiver receiver1() {  **return new** Tut2Receiver(**1**);  }   @Bean  **public** Tut2Receiver receiver2() {  **return new** Tut2Receiver(**2**);  }  }   */\*\*  \* 定义一个消息发送对象  \*  \** ***@return*** *\*/* @Profile(**"sender"**)  @Bean  **public** Tut2Sender sender() {  **return new** Tut2Sender();  } } |

## 发送者

我们将修改发件者，以提供一种方法，通过使用RabbitTemplate的convertAndSend方法发布消息，以非常有创意的方式向邮件中添加点来识别其是否运行更长的任务。convertAndSend即“将Java对象转换为Amqp消息并将其发送到使用默认路由键的默认交换队列中”。

|  |
| --- |
| **package** com.example.rabbitmq;  **import** org.springframework.amqp.core.Queue; **import** org.springframework.amqp.rabbit.core.RabbitTemplate; **import** org.springframework.beans.factory.annotation.Autowired; **import** org.springframework.scheduling.annotation.Scheduled;  */\*\*  \* 消息发送者  \*  \* Author: 王俊超  \* Date: 2017-06-17 10:31  \* All Rights Reserved !!!  \*/* **public class** Tut2Sender {   */\*\* 消息模板对象 \*/* @Autowired  **private** RabbitTemplate template;   */\*\* 消息队列 \*/* @Autowired  **private** Queue **queue**;   **private int dots** = **0**;  **private int count** = **0**;   */\*\*  \* 消息发送方法，初始延迟0.5秒，之后每1秒发送一个消息  \*/* @Scheduled(fixedDelay = **1000**, initialDelay = **500**)  **public void** send() {  StringBuilder builder = **new** StringBuilder(**"Hello"**);  **if** (**dots**++ == **3**) {  **dots** = **1**;  }   **for** (**int** i = **0**; i < **dots**; i++) {  builder.append(**'.'**);  }   builder.append(Integer.*toString*(++**count**));  String message = builder.toString();  template.convertAndSend(**queue**.getName(), message);  System.***out***.println( **" [x] Sent '"** + message + **"'"**);  } } |

## 接收者

我们的接收器Tut2Receiver模拟了doWork（）方法中的假任务的任意长度，其中点数转换为工作所需的秒数。再次，我们在“hello”队列和@RabbitHandler上使用@RabbitListener来接收消息。消费消息的实例被添加到我们的监视器中，以显示哪个实例，消息和处理消息的时间长度。

|  |
| --- |
| **package** com.example.rabbitmq;  **import** org.springframework.amqp.rabbit.annotation.RabbitHandler; **import** org.springframework.amqp.rabbit.annotation.RabbitListener; **import** org.springframework.util.StopWatch;  */\*\*  \* 消息接收者对象，并指定接收的消息队列  \* <p>  \* Author: 王俊超  \* Date: 2017-06-17 10:30  \* All Rights Reserved !!!  \*/* @RabbitListener(queues = **"hello"**) **public class** Tut2Receiver {   **private final int instance**;    **public** Tut2Receiver(**int** i) {  **this**.**instance** = i;  }   */\*\*  \* 指定消息接收后的处理方法，因为发送的消息是string类型，所以接收方法的入参也是string类型  \*  \** ***@param in*** *\** ***@throws*** *InterruptedException  \*/* @RabbitHandler  **public void** receive(String in) **throws** InterruptedException {  StopWatch watch = **new** StopWatch();  watch.start();  System.***out***.println(**"instance "** + **this**.**instance** + **" [x] Received '"** + in + **"'"**);  doWork(in);  System.***out***.println(**"instance "** + **this**.**instance** + **" [x] Done in "** + watch.getTotalTimeSeconds() + **"s"**);  }   **private void** doWork(String in) **throws** InterruptedException {  **for** (**char** ch : in.toCharArray()) {  **if** (ch == **'.'**) {  Thread.*sleep*(**1000**);  }  }  } } |

## 运行

先运行接收者，需要添加运行参数：--spring.profiles.active=hello-world,receiver

再运行发送者，需要添加运行参数：--spring.profiles.active=hello-world,sender

发件者的输出应该如下所示：

|  |
| --- |
| Ready ... running for 10000ms  [x] Sent 'Hello.1'  [x] Sent 'Hello..2'  [x] Sent 'Hello...3'  [x] Sent 'Hello.4'  [x] Sent 'Hello..5'  [x] Sent 'Hello...6'  [x] Sent 'Hello.7'  [x] Sent 'Hello..8'  [x] Sent 'Hello...9'  [x] Sent 'Hello.10' |

而工作进程的输出应该如下所示：

|  |
| --- |
| Ready ... running for 10000ms  instance 1 [x] Received 'Hello.1'  instance 2 [x] Received 'Hello..2'  instance 1 [x] Done in 1.001s  instance 1 [x] Received 'Hello...3'  instance 2 [x] Done in 2.004s  instance 2 [x] Received 'Hello.4'  instance 2 [x] Done in 1.0s  instance 2 [x] Received 'Hello..5' |

## 消息确认

执行任务可能需要几秒钟。你可能会想，如果一个消费者开始一个长期的任务，并且仅仅部分地完成它，就会发生什么。默认情况下，Spring-amqp采用保守的消息确认方式。如果监听器引发异常，容器调用：

|  |
| --- |
| channel.basicReject（deliveryTag，requeue） |

默认情况下，会重新排队，除非您明确设置：

|  |
| --- |
| defaultRequeueRejected = false |

或者监听器抛出一个AmqpRejectAndDontRequeueException。这通常是你从接收者那里获得的一种典型行为。在这种模式下，没有必要担心忘记的确认。处理消息后，侦听器调用：

|  |
| --- |
| channel.basicAck（） |

## 忘记确认

错过basicAck是一个常见的错误，spring-amqp通过默认配置有助于避免这种情况。。这是一个容易的发生的错误，但后果是严重的。当您的客户端退出（可能看起来像随机重新传递）时，消息将被重新传递，但是RabbitMQ将会消耗越来越多的内存，因为它将无法释放任何未包含的消息。

为了调试这种错误，您可以使用rabbitmqctl 打印messages\_unacknowledged字段：

sudo rabbitmqctl list\_queues name message\_ready messages\_unacknowledgedged

在Windows上，删除sudo：

rabbitmqctl.bat list\_queues name message\_ready messages\_unacknowledgedged

## 消息持久化

我们已经学会了如何确保即使消费者死亡，任务也不会丢失。但是如果RabbitMQ服务器停止，我们的任务仍然会丢失。

当RabbitMQ退出或崩溃时，它会忘记队列和消息，除非你不告诉它。需要两件事来确保消息不会丢失：我们需要将队列和消息标记为持久。

首先，我们需要确保RabbitMQ不会失去我们的队列。为了这样做，我们需要将其声明为持久的：

|  |
| --- |
| boolean durable = true ;  channel.queueDeclare（“hello”，durable，false，false，null）; |

虽然这个命令本身是正确的，但是在我们目前的设置中是不行的。这是因为我们已经定义了一个非持久化的名为hello的队列。RabbitMQ不允许您重新定义具有不同参数的现有队列，并会向尝试执行此操作的任何程序返回错误。但是有一个快速的解决方法 - 让我们用不同的名称声明一个队列，例如task\_queue：

|  |
| --- |
| boolean durable = true;  channel.queueDeclare("task\_queue", durable, false, false, null); |

这个queueDeclare更改需要应用于生产者和消费者代码。

在这一点上，我们确信，即使RabbitMQ重新启动，task\_queue队列也不会丢失。现在我们需要通过将MessageProperties（实现BasicProperties）设置为值PERSISTENT\_TEXT\_PLAIN来标记我们的消息。

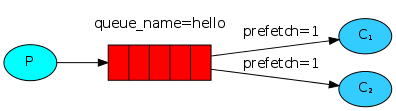
## 注意消息持久性

将消息标记为持久性不能完全保证消息不会丢失。虽然它告诉RabbitMQ将消息保存到磁盘，但是当RabbitMQ接受消息并且尚未保存消息时，仍然有一个很短的时间窗口。此外，RabbitMQ不会对每个消息执行fsync（2） - 它可能只是保存到缓存中，而不是真正写入磁盘。持久性保证不强，但对我们的简单任务队列来说已经足够了。如果您需要更强大的保证，那么您可以使用发布者确认。

## 公平调度

您可能已经注意到，调度仍然无法正常工作。例如在两个工人的情况下，当所有奇怪的信息都很重，甚至信息很轻的时候，一个工作人员将不断忙碌，另一个工作人员几乎不会做任何工作。那么，RabbitMQ不知道什么，还会平均分配消息。

这是因为当消息进入队列时，RabbitMQ只会分派消息。它不看消费者的未确认消息的数量。它只是盲目地向第n个消费者发送每个第n个消息。



为了打破这种方式，我们可以使用basicQos方法与 prefetchCount = 1设置。这告诉RabbitMQ不要一次给一个工作者多个消息。或者换句话说，在处理并确认前一个消息之前，不要向工作进程发送新消息。相反，它将发送到下一个还不忙的工作进程。

|  |
| --- |
| int prefetchCount = 1 ;  channel.basicQos（prefetchCount）; |

## 注意队列大小

如果所有的工人都忙，你的队列可以填满。你会想要注意的是，也许增加更多的工人，或者有其他的策略

## 完整代码

我们的NewTask.java类的最终代码：

|  |
| --- |
| **package** com.example.rabbitmq;  **import** com.rabbitmq.client.Channel; **import** com.rabbitmq.client.Connection; **import** com.rabbitmq.client.ConnectionFactory; **import** com.rabbitmq.client.MessageProperties;  **import** java.io.IOException; **import** java.util.concurrent.TimeoutException;  */\*\*  \* Author: 王俊超  \* Date: 2017-06-09 08:08  \* All Rights Reserved !!!  \*/* **public class** NewTask {  **private static final** String ***TASK\_QUEUE\_NAME*** = **"task\_queue"**;   **public static void** main(String[] args) **throws** IOException, TimeoutException {  ConnectionFactory factory = **new** ConnectionFactory();  factory.setHost(**"localhost"**);  Connection connection = factory.newConnection();  Channel channel = connection.createChannel();   channel.queueDeclare(***TASK\_QUEUE\_NAME***, **true**, **false**, **false**, **null**);   String message = *getMessage*(args);   channel.basicPublish(**""**, ***TASK\_QUEUE\_NAME***,  MessageProperties.***PERSISTENT\_TEXT\_PLAIN***,  message.getBytes(**"UTF-8"**));  System.***out***.println(**" [x] Sent '"** + message + **"'"**);   channel.close();  connection.close();  }   **private static** String getMessage(String[] strings) {  **if** (strings.**length** < **1**) {  **return "Hello World!"**;  }  **return** *joinStrings*(strings, **" "**);  }   **private static** String joinStrings(String[] strings, String delimiter) {  **int** length = strings.**length**;  **if** (length == **0**) {  **return ""**;  }  StringBuilder words = **new** StringBuilder(strings[**0**]);  **for** (**int** i = **1**; i < length; i++) {  words.append(delimiter).append(strings[i]);  }  **return** words.toString();  } } |

和我们的Worker.java：

|  |
| --- |
| **package** com.example.rabbitmq;  **import** com.rabbitmq.client.\*;  **import** java.io.IOException; **import** java.util.concurrent.TimeoutException;  */\*\*  \* Author: 王俊超  \* Date: 2017-06-09 08:02  \* All Rights Reserved !!!  \*/* **public class** Worker {  **public static final** String ***TASK\_QUEUE\_NAME*** = **"task\_queue"**;   **public static void** main(String[] args) **throws** IOException, TimeoutException {  ConnectionFactory factory = **new** ConnectionFactory();  factory.setHost(**"localhost"**);  **final** Connection connection = factory.newConnection();  **final** Channel channel = connection.createChannel();   channel.queueDeclare(***TASK\_QUEUE\_NAME***, **true**, **false**, **false**, **null**);  System.***out***.println(**" [\*] Waiting for messages. To exit press CTRL+C"**);   *// 指定从消息通道中每次取的消息数量* channel.basicQos(**1**);   **final** Consumer consumer = **new** DefaultConsumer(channel) {  @Override  **public void** handleDelivery(String consumerTag, Envelope envelope,  AMQP.BasicProperties properties, **byte**[] body) **throws** IOException {  String message = **new** String(body, **"UTF-8"**);   System.***out***.println(**" [x] Received '"** + message + **"'"**);  **try** {  *doWork*(message);  } **finally** {  System.***out***.println(**" [x] Done"**);  channel.basicAck(envelope.getDeliveryTag(), **false**);  }  }  };   channel.basicConsume(***TASK\_QUEUE\_NAME***, **false**, consumer);  }   **private static void** doWork(String task) {  **for** (**char** ch : task.toCharArray()) {  **if** (ch == **'.'**) {  **try** {  Thread.*sleep*(**1000**);  } **catch** (InterruptedException \_ignored) {  Thread.*currentThread*().interrupt();  }  }  }  } } |

使用消息确认和prefetchCount可以设置工作队列。即使RabbitMQ重新启动，耐久性选项也让任务生存下去。