**声明：此文档为个人翻译官方文档，未经允许请勿外传**

**翻译者：杨德军**

Work Queues

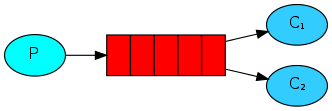
**(using the spring-amqp client)**

**前置条件**

本教程假定RabbitMQ在默认port（5672）上的本地主机上安装并运行。 如果您使用不同的主host, port or credentials，则connections设置需要进行调整。

**从哪里获得帮助**

如果您在阅读本教程时遇到困难，可以通过邮件列表与我们联系([contact us](https://groups.google.com/forum/#!forum/rabbitmq-users))。



在第一篇教程( [first tutorial](http://www.rabbitmq.com/tutorials/tutorial-one-spring-amqp.html))中，我们编写了用于从named queue发送和接收消息的程序。 在这一个中，我们将创建一个*Work Queue*，用于在多个工作workers之间分配耗时的任务。

Work Queues（又称为:*Task Queues*）背后的主要思想是避免立即执行资源密集型(resource-intensive)任务，并且必须等待它完成。 相反，我们安排稍后完成任务。 我们将任务封装为message并将其发送到queue。 在后台运行的工作进程将pop任务并最终执行job。 当你运行许多workers时，任务将在他们之间共享。

这个概念在Web应用程序中特别有用，因为在短的HTTP请求窗口中无法处理复杂的任务。

**准备**

In the previous part of this tutorial we sent a message containing "Hello World!". Now we'll be sending strings that stand for complex tasks. We don't have a real-world task, like images to be resized or pdf files to be rendered, so let's fake it by just pretending we're busy - by using the Thread.sleep() function. We'll take the number of dots in the string as its complexity; every dot will account for one second of "work". For example, a fake task described by Hello... will take three seconds.

Please see the setup in [first tutorial](http://www.rabbitmq.com/tutorials/tutorial-one-spring-amqp.html) if you have not setup the project. We will follow the same pattern as in the first tutorial: 1) create a package (tut2) and create a Tut2Config, Tut2Receiver, and Tut2Sender. Start by creating a new package (tut2) where we'll place our three classes. In the configuration class we setup two profiles, the label for the tutorial ("tut2") and the name of the pattern ("work-queues"). We leverage spring to expose the queue as a bean. We setup the receiver as a profile and define two beans to correspond to the workers in our diagram above; receiver1 and receiver2. Finally, we define a profile for the sender and define the sender bean. The configuration is now done.

import org.springframework.amqp.core.Queue;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.Profile;

@Profile({"tut2", "work-queues"})

@Configuration

public class Tut2Config {

@Bean

public Queue hello() {

return new Queue("hello");

}

@Profile("receiver")

private static class ReceiverConfig {

@Bean

public Tut2Receiver receiver1() {

return new Tut2Receiver(1);

}

@Bean

public Tut2Receiver receiver2() {

return new Tut2Receiver(2);

}

}

@Profile("sender")

@Bean

public Tut2Sender sender() {

return new Tut2Sender();

}

}

**Sender**

We will modify the sender to provide a means for identifying whether its a longer running task by appending a dot to the message in a very contrived fashion using the same method on the RabbitTemplate to publish the message, convertAndSend. The documentation defines this as, "Convert a Java object to an Amqp Message and send it to a default exchange with a default routing key."

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;

public class Tut2Sender {

@Autowired

private RabbitTemplate template;

@Autowired

private Queue queue;

int dots = 0;

int count = 0;

@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 + "'");

}

}

**Receiver**

Our receiver, Tut2Receiver, simulates an arbitary length for a fake task in the doWork() method where the number of dots translates into the number of seconds the work will take. Again, we leverage a @RabbitListener on the "hello" queue and a @RabbitHandler to receive the message. The instance that is consuming the message is added to our monitor to show which instance, the message and the length of time to process the message.

import org.springframework.amqp.rabbit.annotation.RabbitHandler;

import org.springframework.amqp.rabbit.annotation.RabbitListener;

import org.springframework.util.StopWatch;

@RabbitListener(queues = "hello")

public class Tut2Receiver {

private final int instance;

public Tut2Receiver(int i) {

this.instance = i;

}

@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);

watch.stop();

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);

}

}

}

}

**Putting it all together**

Compile them using mvn package and run with the following options

mvn clean package

java -jar target/rabbitmq-amqp-tutorials-0.0.1-SNAPSHOT.jar --spring.profiles.active=work-queues,receiver

java -jar target/rabbitmq-amqp-tutorials-0.0.1-SNAPSHOT.jar --spring.profiles.active=work-queues,sender

The output of the sender should look something like:

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'

And the output from the workers should look something like:

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'

**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. Spring-amqp by default takes a conservative approach to message acknowledgement. If the listener throws an exception the container calls:

channel.basicReject(deliveryTag, requeue)

Requeue is true by default unless you explicitly set:

defaultRequeueRejected=false

or the listener throws an AmqpRejectAndDontRequeueException. This is typically the bahavior you want from your listener. In this mode there is no need to worry about a forgotten acknowledgement. After processing the message the listener calls:

channel.basicAck()

**Forgotten acknowledgment**

It's a common mistake to miss the basicAck and spring-amqp helps to avoid this through its default configuraiton. The consequences are serious. Messages will be redelivered when your client quits (which may look like random redelivery), but RabbitMQ will eat more and more memory as it won't be able to release any unacked messages.

In order to debug this kind of mistake you can use rabbitmqctl to print the messages\_unacknowledged field:

sudo rabbitmqctl list\_queues name messages\_ready messages\_unacknowledged

On Windows, drop the sudo:

rabbitmqctl.bat list\_queues name messages\_ready messages\_unacknowledged

**Message durability**

With spring-amqp there are reasonable default values in the MessageProperties that account for message durability. In particular you can check the table for [common properties](http://docs.spring.io/spring-amqp/reference/htmlsingle/#_common_properties) You'll see two relevant to our discussion here on durability:

| Property | default | Description |
| --- | --- | --- |
| durable | true | When declareExchange is true the durable flag is set to this value |
| deliveryMode | PERSISTENT | PERSISTENT or NON\_PERSISTENT to determine whether or not RabbitMQ should persist the messages |

**Note on message persistence**

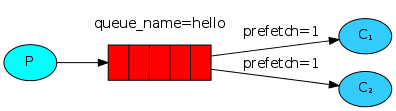
Marking messages as persistent doesn't fully guarantee that a message won't be lost. Although it tells RabbitMQ to save the message to disk, there is still a short time window when RabbitMQ has accepted a message and hasn't saved it yet. Also, RabbitMQ doesn't do fsync(2) for every message -- it may be just saved to cache and not really written to the disk. The persistence guarantees aren't strong, but it's more than enough for our simple task queue. If you need a stronger guarantee then you can use [publisher confirms](https://www.rabbitmq.com/confirms.html).

**Fair dispatch vs Round-robin dispatching**

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. In this mode dispatching doesn't necessarily work exactly as we want. For example in a situation with two workers, when all odd messages are heavy and even messages are light, one worker will be constantly busy and the other one will do hardly any work. Well, RabbitMQ doesn't know anything about that and will still dispatch messages evenly.

This happens because RabbitMQ just dispatches a message when the message enters the queue. It doesn't look at the number of unacknowledged messages for a consumer. It just blindly dispatches every n-th message to the n-th consumer.

However, "Fair dispatch" is the default configuration for spring-amqp. The SimpleMessageListenerContainer defines the value for DEFAULT\_PREFETCH\_COUNT to be 1. If the DEFAULT\_PREFECTH\_COUNT were set to 0 the behavior would be round robin messaging as described above.



However, with the prefetchCount set to 1 by default, this tells RabbitMQ not to give more than one message to a worker at a time. Or, in other words, don't dispatcha new message to a worker until it has processed and acknowledged the previous one. Instead, it will dispatch it to the next worker that is not still busy.

**Note about queue size**

If all the workers are busy, your queue can fill up. You will want to keep an eye on that, and maybe add more workers, or have some other strategy.

By using spring-amqp you get reasonable values configured for message acknowledgments and fair dispatching. The default durability for queues and persistence for messages provided by spring-amqp allow let the messages to survive even if RabbitMQ is restarted.

For more information on Channel methods and MessageProperties, you can browse the [javadocs online](http://docs.spring.io/spring-amqp/docs/current/api/index.html?org/springframework/amqp/package-summary.html) For understanding the underlying foundation for spring-amqp you can find the [rabbitmq-java-client](https://rabbitmq.github.io/rabbitmq-java-client/api/current/).

Now we can move on to [tutorial 3](http://www.rabbitmq.com/tutorials/tutorial-three-spring-amqp.html) and learn how to deliver the same message to many consumers.