**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

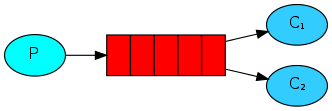
**(using the spring-amqp client)**

**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**

在本教程的前一部分中，我们发送了一条包含“Hello World！”的消息。 现在我们将发送代表复杂任务的字符串。 我们没有real-world task，比如要调整大小的图像或要渲染的PDF文件，所以让我们假装我们很忙 - 使用Thread.sleep()函数来伪装它。 我们将把字符串中的点数作为它的复杂度; 每一个点都会占用一秒的“work”。 例如，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 [first tutorial](http://www.rabbitmq.com/tutorials/tutorial-one-spring-amqp.html)中的设置。 我们将遵循与第一个教程相同的模式：1）创建一个包（tut2）并创建一个Tut2Config，Tut2Receiver和Tut2Sender。 首先创建一个新的包（tut2），我们将在这里放置我们的三个类。 在配置类中，我们设置了两个配置文件，教程标签（“tut2”）和模式名称（“work-queues”）。 我们利用spring将队列公开为bean。 我们将receiver设置为配置文件，并定义两个bean以对应上图中的workers; receiver1和receiver2。 最后，我们为sender定义一个配置文件并定义sender bean。 配置现在完成了。

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**

我们修改sender，以便来确认是否是一个耗时操作，以直观的方式追加点号来说明耗时时长，我们同样适用RabbitTemplate publish message，convertAndSend。该文档将其定义为“将Java对象转换为Amqp message，并使用default routing key 发送到default exchange”。

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**

使用mvn package 编译并使用以下命令运行

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

sender 应该输出如下内容:

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workers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'

**Message acknowledgment(消息确认)**

执行一个task可能需要几秒钟的时间。 你可能想知道如果其中一个consumer开始一项耗时任务并且只完成一部分后死亡会发生什么。 默认情况下，Spring-amqp采用保守的方式来message acknowledgement。 如果listener throws一个container调用的exception：

channel.basicReject(deliveryTag, requeue)

除非你明确设置了以下内容，否则默认为true

defaultRequeueRejected=false

或者listener throws一个AmqpRejectAndDontRequeueExceptionException。 这通常是您想要从listener处获得的bahavior。 在这种模式下，listener处理 message后不需要担心忘记acknowledgement。

channel.basicAck()

**Forgotten acknowledgment(忘记确认)**

没有basicAck是一个常见错误，spring-amqp有助于通过默认配置避免这种情况。 确实basicAck后果是严重的。 当你的client退出时（这可能看起来像随机的重新传送），消息将被重新传递，但是RabbitMQ将会消耗越来越多的内存，因为它将不能释放任何unacked messages。

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

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

在Windows上，删除sudo:

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

**Message durability(消息持久化)**

使用spring-amqp，MessageProperties中有合理的默认值来说明消息的持久性。 特别是你可以检查表格的 [common properties](http://docs.spring.io/spring-amqp/reference/htmlsingle/#_common_properties)你会看到两个与我们在这里讨论的耐久性相关的问题：

| Property | default | Description |
| --- | --- | --- |
| durable | true | declareExchange为true时,durable flag被设置为该值 |
| deliveryMode | PERSISTENT | PERSISTENT or NON\_PERSISTENT 来确定RabbitMQ是否应该persist messages |

**关于message persistence**

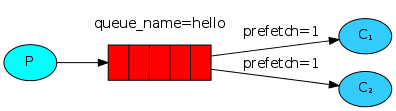
将messages标记为persistentmessages并不能完全保证邮件不会丢失。 尽管它告诉RabbitMQ将消息保存到磁盘，但RabbitMQ接收到消息并且尚未保存消息时仍有一段空窗期。 此外，RabbitMQ不会为每条消息执行fsync(2) - 它可能只是保存到缓存中，并没有真正写入磁盘。 持久性保证不强，但对我们简单的task queue来说已经足够了。 如果您需要更强大的保证，那么您可以使用[publisher confirms](https://www.rabbitmq.com/confirms.html)。

**Fair dispatch vs Round-robin dispatching(公平调度与循环调度)**

默认情况下，RabbitMQ将按顺序将每条消息发送给下一个consumer。 平均而言，每个消费者将获得相同数量的消息。 这种分配消息的方式称为round-robin。 在这种模式下，调度并不一定像我们想要的那样工作。 例如，在有两个workers的情况下，当所有odd messages都很heavy，甚至信息很少时，一个worker会一直很忙，另一个worker几乎不会做任何工作。 那么，RabbitMQ不知道任何有关worker的信息，并仍将均匀地发送消息。

发生这种情况是因为RabbitMQ只在message进入queue时调度消息分发机制。 它没有考虑consumer unacknowledged messages的数量。 它只是盲目地将n-th message to the n-th consumer。

然而，"Fair dispatch"是spring-amqp的默认配置。 SimpleMessageListenerContainer将DEFAULT\_PREFETCH\_COUNT的值定义为1.如果DEFAULT\_PREFECTH\_COUNT设置为0，则行为将如上所述为循环消息(round robin)传递。



但是，默认情况下，prefetchCount设置为1，这告诉RabbitMQ一次不要向worker发送多条消息。 或者换句话说，在处理并确认前一个worker之前，不要向worker发送新消息。 相反，它会将其分派给不忙碌的下一个worker。

**关于queue size**

如果所有的workers都很忙，你的queue可以填满。 你会想看看，也许会增加更多的worker，或者有其他的策略。

通过使用spring-amqp，您可以获得为message acknowledgments和fair dispatching配置的合理值。 queues的default durability和spring-amqp提供的messages的持久性允许即使RabbitMQ重新启动也能保留消息。

有关Channel方法和MessageProperties的更多信息，您可以在线浏览 [javadocs online](http://docs.spring.io/spring-amqp/docs/current/api/index.html?org/springframework/amqp/package-summary.html)。要了解spring-amqp的基础，您可以找到 [rabbitmq-java-client](https://rabbitmq.github.io/rabbitmq-java-client/api/current/)。

现在我们可以继续阅读[tutorial 3](http://www.rabbitmq.com/tutorials/tutorial-three-spring-amqp.html) 并学习如何向许多consumers deliver相同的消息。