Tutorial Original em (inglês): https://www.rabbitmq.com/tutorials/tutorial-two-java.html

Para o código abaixo, você pode usar como base o https://github.com/gugawag/rabbitmq-basico

Filas de trabalho (Work Queues)

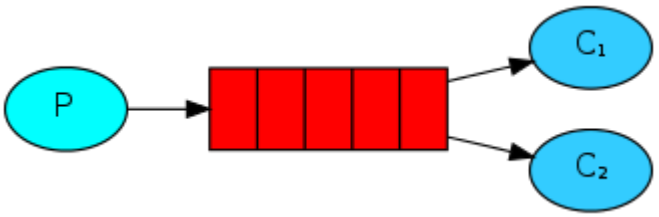
(usando o cliente Java)

Pré-requisitos

Este tutorial assume que o RabbitMQ esteja instalado e em execução no `localhost` na porta padrão (`5672`). Caso você use um host, porta ou credencial diferente, as configurações de conexão exigirão ajustes.

Onde obter ajuda

Se estiver com problemas para passar por este tutorial, entre em contato conosco através da lista de discussão.



No primeiro tutorial, escrevemos programas para enviar e receber mensagens de uma fila nomeada. Nesta, criaremos uma *Fila de trabalho* que será usada para distribuir tarefas demoradas entre vários trabalhadores.

A principal ideia por trás das Filas de trabalho (também conhecida como *Filas de tarefas - Task Queues*) é evitar executar uma *tarefa que consome muitos* recursos imediatamente e ter que esperar a conclusão. Em vez disso, agendamos a tarefa a ser executada mais tarde. Encapsulamos uma *tarefa* como uma mensagem e a enviamos para uma fila. Um *worker (trabalhador)* em execução em segundo plano exibirá as tarefas e, eventualmente, executará o trabalho. Quando você executa muitos trabalhadores, as tarefas serão compartilhadas entre eles.

Esse conceito é especialmente útil em aplicativos da Web onde é impossível lidar com uma tarefa complexa durante uma pequena janela de solicitação HTTP.

Preparação

Na parte anterior deste tutorial, enviamos uma mensagem contendo "Hello World!". Agora estaremos enviando strings que representam tarefas complexas. Não temos uma tarefa do mundo real, como imagens a serem redimensionadas ou arquivos PDF a serem renderizados, então vamos fingir apenas que estamos ocupados - usando a função `Thread.sleep ()` . Levaremos em conta que o número de pontos na string representa sua complexidade; cada ponto será responsável por um segundo de "trabalho". Por exemplo, uma tarefa falsa descrita por `Hello ...` levará três segundos.

Modificaremos levemente o código *Send.java* do nosso exemplo anterior, para permitir que mensagens arbitrárias sejam enviadas a partir da linha de comando. Este programa agendará tarefas para nossa fila de trabalho, então vamos chamá-lo de `NewTask.java` :

```
String mensagem = String.join ("", argv);

channel.basicPublish ("", "olá", null, message.getBytes ());
System.out.println ("[x] Enviado '" + mensagem + "'");
```

Nosso antigo programa *Recv.java* também requer algumas alterações: ele precisa falsificar um segundo de trabalho para cada ponto no corpo da mensagem. Ele manipulará as mensagens entregues e executará a tarefa, então vamos chamá-lo de `Worker.java` :

```
DeliverCallback deliverCallback = (consumerTag, delivery) -> {
  String mensagem = new String (delivery.getBody (), "UTF-8");

  System.out.println ("[x] Recebido '" + mensagem + "'");
  try {
    doWork (mensagem);
  } finally {
    System.out.println ("[x] Feito");
  }
};
boolean autoAck = true; // ack é feito aqui. Como está autoAck, enviará automaticamente
channel.basicConsume (TASK_QUEUE_NAME, autoAck, deliverCallback, consumerTag ->
{});
```

Nossa tarefa falsa para simular o tempo de execução:

```
private static void doWork (String task) throws InterruptedException {
```

```
    for (char ch: task.toCharArray ()) {
        if (ch == '.') Thread.sleep (1000);
    }
}
```

Compile-os como no tutorial um (com os arquivos jar no diretório ativo e a variável de ambiente `CP`):

```
javac -cp $CP NewTask.java Worker.java
```

Expedição round-robin

Uma das vantagens do uso de uma fila de tarefas é a capacidade de paralelizar facilmente o trabalho. Se estamos construindo uma lista de pendências de trabalho, podemos apenas adicionar mais trabalhadores e, dessa forma, escalar facilmente.

Primeiro, vamos tentar executar duas instâncias de trabalho ao mesmo tempo. Ambos receberão mensagens da fila, mas como exatamente? Vamos ver.

Você precisa de três consoles abertos. Dois executarão o programa de trabalho. Esses consoles serão nossos dois consumidores - C1 e C2.

```
# shell 1
java -cp $CP Worker
# => [*] Aguardando mensagens. Para sair, pressione CTRL + C
```

```
# shell 2
java -cp $CP Worker
# => [*] Aguardando mensagens. Para sair, pressione CTRL + C
```

No terceiro, publicaremos novas tarefas. Depois de iniciar os consumidores, você pode publicar algumas mensagens:

```
# shell 3
java -cp $ CP NewTask Primeira mensagem.
# => [x] Enviada 'Primeira mensagem'.
java -cp $ CP NewTask Segunda mensagem ..
# => [x] Enviada 'Segunda mensagem ..'
java -cp $ CP NewTask Terceira mensagem ...
# => [x] Enviada 'Terceira mensagem ...'
```

```
java -cp $ CP NewTask Quarta mensagem ....
# => [x] Enviada 'Quarta mensagem ....'
java -cp $ CP NewTask Quinta mensagem .....
# => [x] Enviada 'Quinta mensagem .....'
```

Vamos ver o que é entregue aos nossos trabalhadores:

```
Trabalhador java -cp $ CP
# => [*] Aguardando mensagens. Para sair, pressione CTRL + C
# => [x] Recebida 'Primeira mensagem'.
# => [x] Recebeu 'terceira mensagem ...'
# => [x] Recebeu 'Quinta mensagem .....'
```

```
Trabalhador java -cp $ CP
# => [*] Aguardando mensagens. Para sair, pressione CTRL + C
# => [x] Recebida 'Segunda mensagem ..'
# => [x] Recebeu 'Quarta mensagem ....'
```

Por padrão, o RabbitMQ enviará cada mensagem para o próximo consumidor, em sequência. Em média, todo consumidor recebe o mesmo número de mensagens. Essa maneira de distribuir mensagens é chamada round-robin. Tente isso com três ou mais trabalhadores.

Confirmação de mensagem (ack)

A execução de uma tarefa pode demorar alguns segundos. Você pode se perguntar **o que acontece se um dos consumidores inicia uma tarefa longa e morre com ela apenas parcialmente concluída**. Com o nosso código atual, uma vez que o RabbitMQ entrega uma mensagem ao consumidor, ele a marca imediatamente para exclusão. Nesse caso, se você matar um trabalhador, perderemos a mensagem que estava sendo processada. Também perderemos todas as mensagens enviadas para esse trabalhador em particular, mas que ainda não foram tratadas.

Mas não queremos perder nenhuma tarefa. Se um trabalhador morrer, gostaríamos que a tarefa fosse entregue a outro trabalhador.

Para garantir que uma mensagem nunca seja perdida, o RabbitMQ suporta *reconhecimentos de* mensagem (message acknowledgements). Uma confirmação (ack) é enviada de volta pelo consumidor para informar ao RabbitMQ que uma mensagem específica foi recebida, processada e que o RabbitMQ está livre para excluí-la.

Se um consumidor morre (seu canal está fechado, a conexão está fechada ou a conexão TCP é perdida) sem enviar uma confirmação, o RabbitMQ entenderá que uma mensagem não foi totalmente processada e a colocará em fila novamente. Se houver outros consumidores on-line ao mesmo tempo, ele os entregará rapidamente a outro consumidor. Dessa forma, você pode ter certeza de que nenhuma mensagem será perdida, mesmo que os trabalhadores ocasionalmente morram.

Não há nenhum tempo limite da mensagem (timeout); O RabbitMQ retornará a mensagem quando o consumidor morrer. Tudo bem, mesmo que o processamento de uma mensagem leve muito, muito tempo. **Perceba que, mais uma vez, não há "timeout" de mensagem, até porque a mensagem pode demorar muito a ser processada. O que há é a detecção, pelo RabbitMQ, da morte do trabalhador e, neste momento, coloca novamente a mensagem na fila.**

As confirmações manuais de mensagens estão ativadas por padrão. Nos exemplos anteriores, nós os desativamos explicitamente através da flag `autoAck = true` . É hora de definir essa flag como `false` e enviar uma confirmação adequada do trabalhador, assim que terminarmos uma tarefa.

```
channel.basicQos (1); // aceita apenas uma mensagem unacked (sem ack) de cada vez (veja abaixo)

DeliverCallback deliverCallback = (consumerTag, delivery) -> {
 String mensagem = new String (delivery.getBody (), "UTF-8");

 System.out.println ("[x] Recebido '" + mensagem + "'");
 try {
   doWork (mensagem);
 } finally {
   System.out.println ("[x] Feito");
   channel.basicAck(delivery.getEnvelope(). getDeliveryTag(), false);
 }
};
boolean autoAck = false;
channel.basicConsume(TASK_QUEUE_NAME, autoAck, deliverCallback, consumerTag -> {});
```

Usando esse código, podemos ter certeza de que, mesmo que você mate um trabalhador usando CTRL + C enquanto processava uma mensagem, nada será perdido. Logo após a morte do trabalhador, todas as mensagens não reconhecidas serão devolvidas.

Teste isso agora: envie mensagens, execute dois trabalhadores, mate um, e veja que a mensagem que o trabalhador morto recebeu será tratado pelo primeiro.

A confirmação deve ser enviada no mesmo canal que recebeu a entrega. Tentativas de reconhecer o uso de um canal diferente resultarão em uma exceção de protocolo no nível do canal. Consulte o guia de documentos sobre confirmações para saber mais.

Reconhecimento (ack) esquecido

É um erro comum esquecer o `basicAck`. É um erro fácil de se cometer, mas as consequências são graves. As mensagens serão entregues novamente quando o cliente for encerrado (o que pode parecer uma reentrega aleatória), mas o RabbitMQ consumirá mais e mais memória, pois não poderá liberar mensagens unacked.

Para depurar esse tipo de erro, você pode usar `rabbitmqctl` para imprimir o campo `messages_unacknowledged` :

```
sudo rabbitmqctl list_queues name messages_ready messages_unacknowledged
```

No Windows, remova o sudo:

```
rabbitmqctl.bat list_queues name messages_ready messages_unacknowledged
```

Durabilidade da mensagem

Aprendemos como garantir que, mesmo que o consumidor morra, a tarefa não se perca. Mas nossas tarefas ainda serão perdidas se o servidor RabbitMQ parar.

Quando o RabbitMQ encerra ou trava, ele esquece as filas e as mensagens, a menos que você o solicite. São necessárias duas coisas para garantir que as mensagens não sejam perdidas: precisamos marcar a fila e as mensagens como duráveis.

Primeiro, precisamos garantir que o RabbitMQ nunca perca nossa fila. Para fazer isso, precisamos declarar como *durável* :

```
boolean duravel = true;
channel.queueDeclare ("fila", duravel, false, false, null);
```

Embora este comando esteja correto por si só, ele não funcionará em nossa configuração atual. Isso porque já definimos uma fila chamada `hello` que não é durável. O RabbitMQ não permite que você redefina uma fila existente com parâmetros diferentes e retornará um erro a qualquer programa que tentar fazer isso. Mas há uma solução rápida - vamos declarar uma fila com nome diferente, por exemplo `task_queue` :

```
boolean duravel = true;
channel.queueDeclare ("task_queue", duravel, falso, falso, nulo);
```

Essa alteração da `queueDeclare` precisa ser aplicada ao código do produtor e do consumidor pois, como já vimos, tanto produtor como consumidor precisam ter a certeza que estão trabalhando com uma fila com as mesmas propriedades esperadas.

Neste ponto, temos certeza de que a fila `task_queue` não será perdida, mesmo se o RabbitMQ reiniciar. Agora precisamos marcar nossas mensagens como persistentes - configurando `MessageProperties` (que implementa `BasicProperties`) com o valor `PERSISTENT_TEXT_PLAIN`.

```
import com.rabbitmq.client.MessageProperties;

channel.basicPublish ("", "task_queue",
        MessageProperties.PERSISTENT_TEXT_PLAIN,
        message.getBytes());
```

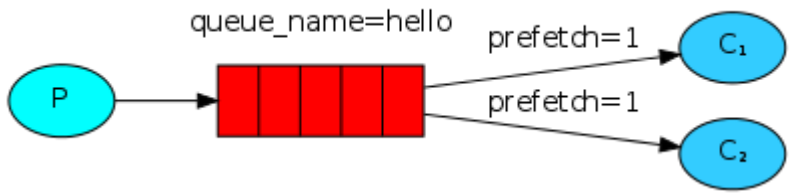
Nota sobre persistência da mensagem

Marcar mensagens como persistentes não garante totalmente que uma mensagem não será perdida. Embora ele diga ao RabbitMQ para salvar a mensagem no disco, ainda há um curto período de tempo em que o RabbitMQ aceitou uma mensagem e ainda não a salvou. Além disso, o RabbitMQ não executa o `fsync (2)` para todas as mensagens - ele pode ser salvo no cache e não gravado no disco. As garantias de persistência não são fortes, mas são mais que suficientes para nossa fila de tarefas simples. Se você precisar de uma garantia mais forte, poderá usar as confirmações do publisher (publisher confirms) .

Envio justo

Você deve ter notado que o envio ainda não funciona exatamente como queremos. Por exemplo, em uma situação com dois trabalhadores, quando todas as mensagens ímpares são pesadas e as mensagens pares são leves, um trabalhador estará constantemente ocupado e o outro quase não fará nenhum trabalho. Bem, o RabbitMQ não sabe nada sobre isso e ainda enviará mensagens uniformemente (da forma Round-Robin).

Isso acontece porque o RabbitMQ apenas envia uma mensagem quando a mensagem entra na fila. Ele não analisa o número de mensagens não reconhecidas para um consumidor. Apenas envia cegamente todas as enésimas mensagens ao enésimo consumidor.



Para evitar isso, podemos usar o método `basicQos` com a configuração `prefetchCount` = `1` . Isso diz ao RabbitMQ para não enviar mais de uma mensagem para um trabalhador de cada vez. Ou, em outras palavras, não despache uma nova mensagem para um trabalhador até que ele tenha processado e reconhecido a anterior. Em vez disso, ela será enviada para o próximo trabalhador que ainda não está ocupado.

```
int prefetchCount = 1;
```

```
channel.basicQos(prefetchCount);
```

Nota sobre o tamanho da fila

Se todos os trabalhadores estiverem ocupados, sua fila poderá ser preenchida. Você vai querer ficar de olho nisso e talvez adicionar mais trabalhadores ou ter alguma outra estratégia.

Juntando tudo

Código final da nossa classe `NewTask.java` :

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

 public class NewTask {

  private static final String TASK_QUEUE_NAME = "task_queue";

  public static void main (String [] argv) throws Exception {
    Factory ConnectionFactory = new ConnectionFactory ();
    factory.setHost ("localhost");
    try (Connection connection = factory.newConnection ();
         Channel channel = connection.createChannel ()) {
        channel.queueDeclare (TASK_QUEUE_NAME, true, false, false, null);

        String mensagem = String.join("", argv);

        channel.basicPublish ("", TASK_QUEUE_NAME,
                MessageProperties.PERSISTENT_TEXT_PLAIN,
                message.getBytes("UTF-8"));
        System.out.println ("[x] Enviado '" + mensagem + "'");
    }
  }

 }
```

(Fonte NewTask.java)

E nosso `Worker.java` :

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

 public class Worker {

  private static final String TASK_QUEUE_NAME = "task_queue";

  public static void main (String [] argv) throws Exception {
    Factory ConnectionFactory = 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 ("[*] Aguardando mensagens. Para sair, pressione CTRL + C");

    channel.basicQos(1);

    DeliverCallback deliverCallback = (consumerTag, delivery) -> {
        String mensagem = new String (delivery.getBody (), "UTF-8");

        System.out.println ("[x] Recebido '" + mensagem + "'");
        try {
            doWork (mensagem);
        } finally {
            System.out.println ("[x] Done");
            channel.basicAck (delivery.getEnvelope (). getDeliveryTag (), false);
        }
    };
    channel.basicConsume (TASK_QUEUE_NAME, false, deliverCallback, consumerTag ->
{});
  }

  private static void doWork(String task) {
    for (char ch: task.toCharArray ()) {
        if (ch == '.') {
            try {
                Thread.sleep (1000);
            } catch (InterruptedException _ignored) {
                Thread.currentThread().interrupt();
            }
        }
    }
  }
```

```
}
```

(Origem Worker.java)

Usando confirmações de mensagem e `prefetchCount`, você pode configurar uma fila de trabalho. As opções de durabilidade permitem que as tarefas sobrevivam mesmo que o RabbitMQ seja reiniciado.

Para obter mais informações sobre `Channel` e `MessageProperties`, você pode navegar pelos JavaDocs online.

O que entregar?

1. Altere o código para enviar uma mensagem com seu nome completo no meio;
2. Dê um printscreen da execução com seu nome (contendo o código-fonte alterado)
3. Coloque seu código no seu github e envie aqui o link do projeto.