# 11-rabbitmq-远程过程调用

## 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。

## 远程过程调用

在第二个教程中，我们学习了如何使用工作队列在多个工作人员之间分配耗时的任务。

但是，如果我们需要在远程计算机上运行功能并等待结果怎么办？那是一个不同的故事。此模式通常称为远程过程调用或RPC。

在本教程中，我们将使用RabbitMQ构建一个RPC系统：一个客户机和一个可扩展的RPC服务器。由于我们没有任何值得分发的耗时任务，我们将创建一个返回斐波纳契数字的虚拟RPC服务。

## 客户端界面

为了说明如何使用RPC服务，我们将创建一个简单的客户端类。它将公开一个名为call的方法，该方法 发送RPC请求并阻塞，直到接收到答案：

|  |
| --- |
| FibonacciRpcClient fibonacciRpc = new FibonacciRpcClient();  String result = fibonacciRpc.call("4");  System.out.println( "fib(4) is " + result); |

## RPC注意事项

虽然RPC是一个很常见的计算模式，但它经常被批评。当程序员不知道函数调用是本地函数还是缓慢的RPC时，出现问题。这样的混乱导致了一个不可预测的系统，并增加了调试的不必要的复杂性。而不是简化软件，滥用RPC可能导致不可维护的意大利面条代码。

铭记这一点，请考虑以下建议：

* 确保显而易见哪个函数调用是本地的，哪个是远程的。
* 记录您的系统。使组件之间的依赖关系清除。
* 处理错误情况。当RPC服务器停机很长时间后，客户端应该如何反应？

当有疑问避免RPC。如果可以的话，您应该使用异步管道 - 而不是类似RPC的阻塞，将异步推送到下一个计算阶段。

## 回调队列

一般来说RPC对RabbitMQ来说很容易。客户端发送请求消息，服务器回复一条响应消息。为了收到响应，我们需要发送一个'回调'队列地址与请求。我们可以使用默认队列（在Java客户端中是排他性的）。我们来试试吧

|  |
| --- |
| callbackQueueName = channel.queueDeclare().getQueue();  BasicProperties props = new BasicProperties  .Builder()  .replyTo(callbackQueueName)  .build();  channel.basicPublish("", "rpc\_queue", props, message.getBytes());  // ... then code to read a response message from the callback\_queue ... |

我们需要这个新的导入：

|  |
| --- |
| import com.rabbitmq.client.AMQP.BasicProperties; |

## 消息属性

AMQP 0-9-1协议预先定义了一组14个随附消息的属性。大多数属性很少使用，除了以下内容：

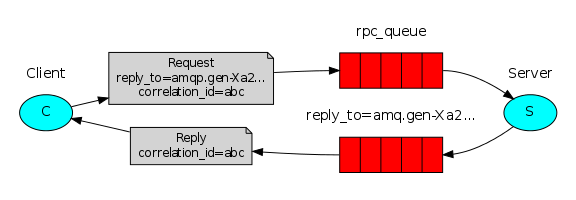
* deliveryMode：将消息标记为persistent（值为2）或transient（任何其他值）。您可能会从第二个教程中记住此属性。
* contentType：用于描述mime类型的编码。例如对于经常使用的JSON编码，将此属性设置为：application / json是一个很好的做法。
* replyTo：通常用来命名一个回调队列。
* correlationId：用于将RPC响应与请求相关联。

## 相关标识（Correlation Id）

在上面提出的方法中，我们建议为每个RPC请求创建一个回调队列。这是非常低效的，但幸运的是有一个更好的方法 - 让我们为每个客户端创建一个回调队列。

这引发了一个新问题，在该队列中收到响应，响应所属的请求不清楚。那就是在使用correlationId属性的时候 。我们将为每个请求设置一个唯一的值。稍后，当我们在回调队列中收到一条消息时，我们将查看此属性，并且基于此，我们将能够将响应与请求相匹配。如果我们看到一个未知的 correlationId值，我们可以安全地丢弃该消息 - 它不属于我们的请求。

## 概要



我们的RPC将像这样工作：

* 当客户端启动时，它创建一个匿名独占回调队列。
* 对于RPC请求，客户端发送一个具有两个属性的消息：replyTo，它被设置为回调队列和correlationId，它被设置为每个请求的唯一值。
* 请求被发送到rpc\_queue队列。
* RPC 工作进程正在等待队列上的请求。当请求出现时，它将执行该作业，并使用replyTo字段中的队列将结果发送回客户端。
* 客户端等待回调队列中的数据。当信息出现时，它检查correlationId属性。如果它与请求中的值相匹配，则返回对应用程序的响应。

## 把它们放在一起

斐波纳契任务：

|  |
| --- |
| private static int fib(int n) {  if (n == 0) return 0;  if (n == 1) return 1;  return fib(n-1) + fib(n-2);  } |

我们声明我们的fibonacci功能。它只假定有效的正整数输入。（不要指望这个工作的大数字，这可能是最慢的递归实现）。

我们的RPC服务器RPCServer.java的代码如下所示：

|  |
| --- |
| **package** com.example.rabbitmq;  **import** com.rabbitmq.client.\*;  **import** java.io.IOException; **import** java.util.concurrent.TimeoutException;  */\*\*  \* Author: 王俊超  \* Date: 2017-06-17 21:41  \* All Rights Reserved !!!  \*/* **public class** RPCServer {  **private static final** String ***RPC\_QUEUE\_NAME*** = **"rpc\_queue"**;   **private static int** fib(**int** n) {  **if** (n <= **0**) {  **return 0**;  }  **if** (n == **1**) {  **return 1**;  }   **return** *fib*(n - **1**) + *fib*(n - **2**);  }   **public static void** main(String[] argv) {  ConnectionFactory factory = **new** ConnectionFactory();  factory.setHost(**"localhost"**);   Connection connection = **null**;  **try** {  connection = factory.newConnection();  Channel channel = connection.createChannel();   channel.queueDeclare(***RPC\_QUEUE\_NAME***, **false**, **false**, **false**, **null**);   channel.basicQos(**1**);   System.***out***.println(**" [x] Awaiting RPC requests"**);   Consumer consumer = **new** DefaultConsumer(channel) {  @Override  **public void** handleDelivery(String consumerTag, Envelope envelope,  AMQP.BasicProperties properties, **byte**[] body) **throws** IOException {  AMQP.BasicProperties replyProps = **new** AMQP.BasicProperties  .Builder()  .correlationId(properties.getCorrelationId())  .build();   String response = **""**;   **try** {  String message = **new** String(body, **"UTF-8"**);  **int** n = Integer.*parseInt*(message);   System.***out***.println(**" [.] fib("** + message + **")"**);  response += *fib*(n);  } **catch** (RuntimeException e) {  System.***out***.println(**" [.] "** + e.toString());  } **finally** {  *// 将执行结果发回给回调队列* channel.basicPublish(**""**, properties.getReplyTo(), replyProps, response.getBytes(**"UTF-8"**));  channel.basicAck(envelope.getDeliveryTag(), **false**);  }  }  };   channel.basicConsume(***RPC\_QUEUE\_NAME***, **false**, consumer);   *//loop to prevent reaching finally block* **while** (**true**) {  **try** {  Thread.*sleep*(**100**);  } **catch** (InterruptedException \_ignore) {  }  }  } **catch** (IOException | TimeoutException e) {  e.printStackTrace();  } **finally** {  **if** (connection != **null**) {  **try** {  connection.close();  } **catch** (IOException \_ignore) {  }  }  }  } } |

服务器代码相当简单：

像往常一样，我们开始建立连接，通道并声明队列。

我们可能想要运行多个服务器进程。为了在多个服务器上平均分配负载，我们需要在channel.basicQos中设置prefetchCount值。

我们使用basicConsume访问队列，我们​​以对象（DefaultConsumer）的形式提供回调，该对象将执行该操作并将响应发送回来。

我们的RPC客户端代码RPCClient.java：

|  |
| --- |
| **package** com.example.rabbitmq;  **import** com.rabbitmq.client.\*;  **import** java.io.IOException; **import** java.util.UUID; **import** java.util.concurrent.ArrayBlockingQueue; **import** java.util.concurrent.BlockingQueue; **import** java.util.concurrent.TimeoutException;  */\*\*  \* Author: 王俊超  \* Date: 2017-06-17 21:28  \* All Rights Reserved !!!  \*/* **public class** RPCClient {  **private** Connection **connection**;  **private** Channel **channel**;  **private** String **requestQueueName** = **"rpc\_queue"**;  **private** String **replyQueueName**;   **public** RPCClient() **throws** IOException, TimeoutException {  ConnectionFactory factory = **new** ConnectionFactory();  factory.setHost(**"localhost"**);  **connection** = factory.newConnection();  **channel** = **connection**.createChannel();  **replyQueueName** = **channel**.queueDeclare().getQueue();  }   **public** String call(String message) **throws** IOException, InterruptedException {  String corrId = UUID.*randomUUID*().toString();  AMQP.BasicProperties props = **new** AMQP.BasicProperties  .Builder()  .correlationId(corrId)  .replyTo(**replyQueueName**)  .build();   *// 向队列中发送消息* **channel**.basicPublish(**""**, **requestQueueName**, props, message.getBytes());   **final** BlockingQueue<String> response = **new** ArrayBlockingQueue<String>(**1**);   *// 消费回调结果* **channel**.basicConsume(**replyQueueName**, **true**, **new** DefaultConsumer(**channel**) {  @Override  **public void** handleDelivery(String consumerTag, Envelope envelope,  AMQP.BasicProperties properties, **byte**[] body) **throws** IOException {  **if** (properties.getCorrelationId().equals(corrId)) {  *// 将消息插入到阻塞队列中* response.offer(**new** String(body, **"UTF-8"**));  }  }  });   *// 从阻塞队列中取消息* **return** response.take();  }   **public void** close() **throws** IOException {  **connection**.close();  }   **public static void** main(String[] argv) **throws** Exception {  RPCClient fibonacciRpc = **null**;  String response = **null**;  fibonacciRpc = **new** RPCClient();  System.***out***.println(**" [x] Requesting fib(30)"**);  response = fibonacciRpc.call(**"30"**);  System.***out***.println(**" [.] Got '"** + response + **"'"**);  fibonacciRpc.close();  }  } |

客户端代码稍微涉及：

我们建立一个连接和通道，并为回复声明一个独占的'回调'队列。

我们订阅'回调'队列，这样我们可以收到RPC响应。

我们的调用方法使得实际的RPC请求。

在这里，我们首先生成一个唯一的correlationId 数字并保存它 - 我们 在DefaultConsumer中实现handleDelivery将使用此值来捕获适当的响应。

接下来，我们发布请求消息，具有两个属性： replyTo和correlationId。

在这一点上，我们可以坐下来等待适当的响应到达。

由于我们的消费者交付处理发生在一个单独的线程中，我们将需要一些东西在响应到达之前暂停主线程。用法的BlockingQueue是可能的解决方案之一。这里我们正在创建ArrayBlockingQueue ，容量设置为1，因为我们只需要等待一个响应。

该handleDelivery方法是做一个很简单的工作，对每一位消费响应消息它会检查的correlationId是我们要找的人。如果是这样，它将对BlockingQueue进行响应。

同时主线程正在等待响应从BlockingQueue取出。

最后，我们将响应返回给用户。

让客户请求：

|  |
| --- |
| RPCClient fibonacciRpc = new RPCClient();  System.out.println(" [x] Requesting fib(30)");  String response = fibonacciRpc.call("30");  System.out.println(" [.] Got '" + response + "'");  fibonacciRpc.close(); |

## 运行

先运行接收者，再运行发送者

## 说明

这里提出的设计不是RPC服务的唯一可能的实现，而是具有一些重要的优点：

* 如果RPC服务器太慢，可以通过运行另一个RPC服务器进行扩展。尝试在新的控制台中运行第二个RPCServer。
* 在客户端，RPC需要发送和接收一条消息。不需要像queueDeclare这样的同步调用 。因此，RPC客户端只需要一个网络往返单个RPC请求。

我们的代码仍然非常简单，不会尝试解决更复杂（但重要的）问题，例如：

* 如果没有服务器运行，客户端应该如何反应？
* 客户端是否需要RPC的某种超时时间？
* 如果服务器发生故障并引发异常，应该将其转发给客户端？
* 在处理之前防止无效的传入消息（例如检查边界，类型）。