**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**

Remote procedure call (RPC)

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

**Prerequisites**

This tutorial assumes RabbitMQ is [installed](http://www.rabbitmq.com/download.html) and running on localhost on standard port (5672). In case you use a different host, port or credentials, connections settings would require adjusting.

**Where to get help**

If you're having trouble going through this tutorial you can [contact us](https://groups.google.com/forum/#!forum/rabbitmq-users) through the mailing list.

In the [second tutorial](http://www.rabbitmq.com/tutorials/tutorial-two-spring-amqp.html) we learned how to use *Work Queues* to distribute time-consuming tasks among multiple workers.

But what if we need to run a function on a remote computer and wait for the result? Well, that's a different story. This pattern is commonly known as *Remote Procedure Call* or *RPC*.

In this tutorial we're going to use RabbitMQ to build an RPC system: a client and a scalable RPC server. As we don't have any time-consuming tasks that are worth distributing, we're going to create a dummy RPC service that returns Fibonacci numbers.

**Client interface**

To illustrate how an RPC service could be used we're going to change the names of our profiles from "Sender" and "Receiver to "Client" and "Server". When we call the server we will get back the fibonacci of the argument we call with.

Integer response = (Integer) template.convertSendAndReceive

(exchange.getName(), "rpc", start++);

System.out.println(" [.] Got '" + response + "'");

**A note on RPC**

Although RPC is a pretty common pattern in computing, it's often criticised. The problems arise when a programmer is not aware whether a function call is local or if it's a slow RPC. Confusions like that result in an unpredictable system and adds unnecessary complexity to debugging. Instead of simplifying software, misused RPC can result in unmaintainable spaghetti code.

Bearing that in mind, consider the following advice:

* Make sure it's obvious which function call is local and which is remote.
* Document your system. Make the dependencies between components clear.
* Handle error cases. How should the client react when the RPC server is down for a long time?

When in doubt avoid RPC. If you can, you should use an asynchronous pipeline - instead of RPC-like blocking, results are asynchronously pushed to a next computation stage.

**Callback queue**

In general doing RPC over RabbitMQ is easy. A client sends a request message and a server replies with a response message. In order to receive a response we need to send a 'callback' queue address with the request. Spring-amqp's RabbitTemplate handles the callback queue for us when we use the above 'convertSendAndReceive()' method. There is no need to do any other setup when using the RabbitTemplate. For a thorough explanation please see [Request/Reply Message](http://docs.spring.io/spring-amqp/reference/htmlsingle/#request-reply).

**Message properties**

The AMQP 0-9-1 protocol predefines a set of 14 properties that go with a message. Most of the properties are rarely used, with the exception of the following:

* deliveryMode: Marks a message as persistent (with a value of 2) or transient (any other value). You may remember this property from [the second tutorial](http://www.rabbitmq.com/tutorials/tutorial-two-spring-amqp.html).
* contentType: Used to describe the mime-type of the encoding. For example for the often used JSON encoding it is a good practice to set this property to: application/json.
* replyTo: Commonly used to name a callback queue.
* correlationId: Useful to correlate RPC responses with requests.

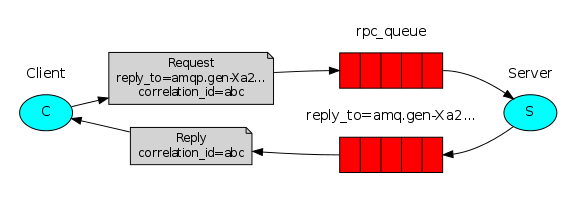
**Correlation Id**

Spring-amqp allows you to focus on the message style you're working with and hide the details of message plumbing required to support this style. For example, typically the native client would create a callback queue for every RPC request. That's pretty inefficient so an alternative is to create a single callback queue per client.

That raises a new issue, having received a response in that queue it's not clear to which request the response belongs. That's when the correlationId property is used. Spring-amqp automatically sets a unique value for every request. In addition it handles the details of matching the response with the correct correlationID.

One reason that spring-amqp makes rpc style easier is that sometimes you may want to ignore unknown messages in the callback queue, rather than failing with an error. It's due to a possibility of a race condition on the server side. Although unlikely, it is possible that the RPC server will die just after sending us the answer, but before sending an acknowledgment message for the request. If that happens, the restarted RPC server will process the request again. The spring-amqp client handles the duplicate responses gracefully, and the RPC should ideally be idempotent.

**Summary**



Our RPC will work like this:

* The Tut6Config will setup a new DirectExchange and a client
* The client will leverage the convertSendAndReceive passing the exchange name, the routingKey, and the message.
* The request is sent to an rpc\_queue ("tut.rpc") queue.
* The RPC worker (aka: server) is waiting for requests on that queue. When a request appears, it performs the task and sends a message with the result back to the Client, using the queue from the replyTo field.
* The client waits for data on the callback queue. When a message appears, it checks the correlationId property. If it matches the value from the request it returns the response to the application. Again, this is done automagically via the RabbitTemplate.

Putting it all together

The Fibonacci task is a @RabbitListener and is defined as:

public int fib(int n) {

return n == 0 ? 0 : n == 1 ? 1 : (fib(n - 1) + fib(n - 2));

}

We declare our fibonacci function. It assumes only valid positive integer input. (Don't expect this one to work for big numbers, and it's probably the slowest recursive implementation possible).

The code for our Tut6Config [Tut6Config](https://github.com/rabbitmq/rabbitmq-tutorials/blob/master/spring-amqp/src/main/java/org/springframework/amqp/tutorials/tut6/Tut6Config.java) looks like this:

import org.springframework.amqp.core.Binding;

import org.springframework.amqp.core.BindingBuilder;

import org.springframework.amqp.core.DirectExchange;

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({"tut6","rpc"})

@Configuration

public class Tut6Config {

@Profile("client")

private static class ClientConfig {

@Bean

public DirectExchange exchange() {

return new DirectExchange("tut.rpc");

}

@Bean

public Tut6Client client() {

return new Tut6Client();

}

}

@Profile("server")

private static class ServerConfig {

@Bean

public Queue queue() {

return new Queue("tut.rpc.requests");

}

@Bean

public DirectExchange exchange() {

return new DirectExchange("tut.rpc");

}

@Bean

public Binding binding(DirectExchange exchange,

Queue queue) {

return BindingBuilder.bind(queue)

.to(exchange)

.with("rpc");

}

@Bean

public Tut6Server server() {

return new Tut6Server();

}

}

}

It setups up our profiles as "tut6" or "rpc". It also setups a "client" profile with two beans; 1) the DirectExchange we are using and 2) the Tut6Client itself. We also configure the "server" profile with three beans, the "tut.rpc.requests" queue, the DirextExchange, which matches the client's exchange, and the binding from the queue to the exchange with the "rpc" routing-key.

The server code is rather straightforward:

* As usual we start annotating our receiver method with a @RabbitListener and defining the queue its listening on.
* Our fibanacci method calls fib() with the payload parameter and returns the result

The code for our RPC client [Tut6Server.java](https://github.com/rabbitmq/rabbitmq-tutorials/blob/master/spring-amqp/src/main/java/org/springframework/amqp/tutorials/tut6/Tut6Server.java):

package org.springframework.amqp.tutorials.tut6;

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

public class Tut6Server {

@RabbitListener(queues = "tut.rpc.requests")

*// @SendTo("tut.rpc.replies") used when the*

*// client doesn't set replyTo.*

public int fibonacci(int n) {

System.out.println(" [x] Received request for " + n);

int result = fib(n);

System.out.println(" [.] Returned " + result);

return result;

}

public int fib(int n) {

return n == 0 ? 0 : n == 1 ? 1 : (fib(n - 1) + fib(n - 2));

}

}

The client code [Tut6Client](https://github.com/rabbitmq/rabbitmq-tutorials/blob/master/spring-amqp/src/main/java/org/springframework/amqp/tutorials/tut6/Tut6Client.java) is as easy as the server:

* We autowire the RabbitTemplate and the DirectExchange bean as defined in the Tut6Config.
* We invoke template.convertSendAndReceive with the parameters exchange name, routing key and message.
* We print the result

Making the Client request is simply:

import org.springframework.amqp.core.DirectExchange;

import org.springframework.amqp.rabbit.core.RabbitTemplate;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.scheduling.annotation.Scheduled;

public class Tut6Client {

@Autowired

private RabbitTemplate template;

@Autowired

private DirectExchange exchange;

int start = 0;

@Scheduled(fixedDelay = 1000, initialDelay = 500)

public void send() {

System.out.println(" [x] Requesting fib(" + start + ")");

Integer response = (Integer) template.convertSendAndReceive

(exchange.getName(), "rpc", start++);

System.out.println(" [.] Got '" + response + "'");

}

}

Using the project setup as defined in (see [tutorial one](http://www.rabbitmq.com/tutorials/tutorial-one-spring-amqp.html)) with start.spring.io and SpringInitialzr the preparing the runtime is the same as the other tutorials:

mvn clean package

We can start the server with:

java -jar target/rabbit-tutorials-1.7.1.RELEASE.jar

--spring.profiles.active=rpc,server

--tutorial.client.duration=6000

To request a fibonacci number run the client:

java -jar target/rabbit-tutorials-1.7.1.RELEASE.jar

--spring.profiles.active=rpc,server

java -jar target/rabbit-tutorials-1.7.1.RELEASE.jar

--spring.profiles.active=rpc,client

The design presented here is not the only possible implementation of a RPC service, but it has some important advantages:

* If the RPC server is too slow, you can scale up by just running another one. Try running a second RPCServer in a new console.
* On the client side, the RPC requires sending and receiving only one message with one method. No synchronous calls like queueDeclare are required. As a result the RPC client needs only one network round trip for a single RPC request.

Our code is still pretty simplistic and doesn't try to solve more complex (but important) problems, like:

* How should the client react if there are no servers running?
* Should a client have some kind of timeout for the RPC?
* If the server malfunctions and raises an exception, should it be forwarded to the client?
* Protecting against invalid incoming messages (eg checking bounds, type) before processing.

If you want to experiment, you may find the [management UI](http://www.rabbitmq.com/management.html) useful for viewing the queues.

There is one other nice feature of RabbitMQ. It is featured as a supported tile on Pivotal Cloud Foundry (PCF) as a service.