# Remote procedure call (RPC)

在第二个教程中，我们学会了如何使用工作队列来分配多个工人之间的耗时的任务。

但是假如我们需要远程计算机上运行，并等待结果呢？

这种模式通常被称为 远程过程调用 或者 RPC

在本教程中我们将使用RabbitMQ搭建一个RPC系统：一个客户端和一个可扩展的RPC服务器。

当我们没有任何费时任务是可以分布的，我们要创建一个虚拟的RPC服务返回斐波那契数列。

## 1、客户端接口

为了说明一个 RPC 服务能被使用，我们将创建一个简单的客户端类，展示名叫 call 的方法，发送 RPC 请求 并直到回复被接收的时候阻塞。

FibonacciRpcClient fibonacciRpc **=** **new** FibonacciRpcClient**();**

String result **=** fibonacciRpc**.**call**(**"4"**);**

System**.**out**.**println**(** "fib(4) is " **+** result**);**

虽然 RPC 在计算机当中是非常常见的模式，但是具有一定的争议。当程序员没有意识到一个功能是否调用了一个局部的或者缓慢的 RPC 时问题就出现了。会出现类似这样的困惑：会导致系统不可预知，和在调试的时候增加了不必要的复杂性。

滥用 RPC 会导致难以维护的代码。

考虑到这，考虑以下建议：

1. 确保调用函数时，明显的知道是局部的还是远程的。
2. 记录你的系统。使组件之间的依赖关系清晰。
3. 处理错误案例。当RPC服务器关闭了很长时间客户端该怎么反应

## 2、Callback queue

通常来说，通过 RabbitMQ 做 RPC 是很容易的。客户端发送一个请求消息，服务器回复一个响应消息。

为了接受到响应，我们发送的一个 ‘callback’ 队列地址的请求。我们可以使用默认队列(这个在客户端专属)

。

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 ...*

### 2.1 、消息属性

AMQP 协议预定义了一组 14 个属性，会随着一个消息的发送而附带过去。大部分属性都很少使用，除了这几个属性：

1. deliveryMode: 发送模式，标记一个消息作为持久性或短暂的
2. contentType:内容类型，用来描述编码的 MIME 类型。例如：经常使用的 JSON 编码就是设置这个属性的很好的做法： application/json
3. replyTo: 回复，常用来命名一个回调队列
4. correlationId：关联Id。关联 RPC 的响应和请求时用到。

## 3、Correlation Id

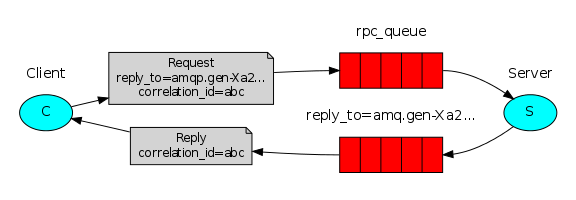
在方法上我们建议为每一个RPC请求回调队列。这是相当低效的，幸运的是有一个更好的方式，每个客户端 都创建一个回调队列(callback queue)。

但是这就有了一个新的问题，该队列接收到一个新的响应，它不清楚这个响应式属于哪个请求的。correlationId 的属性就用在这个时候。我们要将它设置为每一个请求的一个唯一值。稍后，当我们在回调队列中接收到消息时，我们会看到这个属性，在这个基础上我们能够匹配一个请求的响应。如果我们看到一个 未知的 correlationId 我们能够安全地丢弃掉改消息，因为它不属于我们的请求。

为什么我们在回调队列中应该忽略未知的消息，而不是因错误而失败呢？

这是由于服务器端一个竞争条件的可能性。虽然不太可能，但是 RPC服务器在发送确认消息的请求前，回复我们消息之后可能死亡。如果发生这种情况，重启后的 RPC 服务器会再次处理请求，这就是为什么客户端需要优雅地处理重复的消息，和 RPC 最好是幂等的。

## 4、概述



我们的 RPC 会是这样的：

1. 当客户端启动时，它创建一个匿名的唯一的回调队列(callback queue)。
2. 对于一个 RPC 请求，客户端发送消息时会附带两个属性：replyTo, 这是设置回调队列, correlationId 为每个请求设置唯一值。
3. 请求被发送到 rpc\_queue 队列
4. RPC的 worker(服务器) 等待队列的消息，当一个请求出现时， 它便开始工作和发送携带结果的消息返回给客户端，从 replyTo 字段使用队列。
5. 客户端在回调队列上等待数据，当消息出现时 它就检查 correlationId 属性，如果与请求当中的值匹配的话，就返回响应给应用程序.

## 5、代码实现

Fibonacci ：

**private** **static** **int** **fib(int** n**)** **throws** Exception **{**

**if** **(**n **==** 0**)** **return** 0**;**

**if** **(**n **==** 1**)** **return** 1**;**

**return** **fib(**n**-**1**)** **+** fib**(**n**-**2**);**

**}**

**import** com.rabbitmq.client.AMQP.BasicProperties**;**

### 5.1 RPCServer

**package** com.symbol.rabbitmq.rpc;

**import** com.rabbitmq.client.AMQP.BasicProperties;

**import** com.rabbitmq.client.Channel;

**import** com.rabbitmq.client.Connection;

**import** com.rabbitmq.client.ConnectionFactory;

**import** com.rabbitmq.client.QueueingConsumer;

**public** **class** RPCServer {

**private** **static** **final** String ***RPC\_QUEUE\_NAME*** = "rpc\_queue";

**public** **static** **void** main(String[] args) **throws** Exception {

//创建连接

ConnectionFactory factory = **new** ConnectionFactory();

factory.setHost("localhost");

Connection connection = factory.newConnection();

//创建通道

Channel channel = connection.createChannel();

//创建并声明队列

channel.queueDeclare(***RPC\_QUEUE\_NAME***, **false**, **false**, **false**, **null**);

//服务端发送信息的最大数量，如果 0 则是没有限制

channel.basicQos(1);

//创建消费者

QueueingConsumer consumer = **new** QueueingConsumer(channel);

//这里关闭了自动确认应答(其作用见在 worker 队列中说明)

channel.basicConsume(***RPC\_QUEUE\_NAME***, **false**, consumer);

System.***out***.println(" [x] Awaiting RPC requests");

**while** (**true**) {

QueueingConsumer.Delivery delivery = consumer.nextDelivery();

//根据 delivery 发送，得到 BasicProperties 属性对象，获取到客户端锁设置的 联合id 属性

BasicProperties props = delivery.getProperties();

//根据联合 id 重新创建一个 BasicProperties 属性对象(为了让回复的响应辨别是哪个请求)

BasicProperties replyProps = **new** BasicProperties.Builder().correlationId(props.getCorrelationId()).build();

String message = **new** String(delivery.getBody());

**int** n = Integer.*parseInt*(message);

System.***out***.println(" [.] fib(" + message + ")");

String response = "" + *fib*(n);

//返回消息给客户端

channel.basicPublish("", props.getReplyTo(), replyProps, response.getBytes());

//处理完消息，手动确认应答

channel.basicAck(delivery.getEnvelope().getDeliveryTag(), **false**);

}

}

**private** **static** **int** fib(**int** n) **throws** Exception {

**if** (n == 0)

**return** 0;

**if** (n == 1)

**return** 1;

**return** *fib*(n - 1) + *fib*(n - 2);

}

}

### 5.2 RPCClient

**package** com.symbol.rabbitmq.rpc;

**import** com.rabbitmq.client.AMQP.BasicProperties;

**import** com.rabbitmq.client.Channel;

**import** com.rabbitmq.client.Connection;

**import** com.rabbitmq.client.ConnectionFactory;

**import** com.rabbitmq.client.QueueingConsumer;

**public** **class** RPCClient {

**private** Connection connection;

**private** Channel channel;

**private** String requestQueueName = "rpc\_queue";

**private** String replyQueueName;

**private** QueueingConsumer consumer;

// new RPC 对象时 初始化

**public** RPCClient() **throws** Exception {

//创建连接

ConnectionFactory factory = **new** ConnectionFactory();

factory.setHost("localhost");

connection = factory.newConnection();

channel = connection.createChannel();

//

replyQueueName = channel.queueDeclare().getQueue();

consumer = **new** QueueingConsumer(channel);

channel.basicConsume(replyQueueName, **true**, consumer);

}

**public** String call(String message) **throws** Exception {

String response = **null**;

//生成随机的 联合Id

String corrId = java.util.UUID.*randomUUID*().toString();

//设置联合id属性

BasicProperties props = **new** BasicProperties.Builder().correlationId(corrId).replyTo(replyQueueName).build();

//发布消息

channel.basicPublish("", requestQueueName, props, message.getBytes());

**while** (**true**) {

QueueingConsumer.Delivery delivery = consumer.nextDelivery();

**if** (delivery.getProperties().getCorrelationId().equals(corrId)) {

response = **new** String(delivery.getBody());

**break**;

}

}

**return** response;

}

**public** **void** close() **throws** Exception {

connection.close();

}

}

}

### 5.3 测试代码

**package** com.symbol.rabbitmq.rpc;

**public** **class** TestRPCMessage {

**public** **static** **void** main(String[] args) **throws** Exception {

RPCClient fibonacciRpc = **new** RPCClient();

System.***out***.println(" [x] Requesting fib(30)");

String response = fibonacciRpc.call("30");

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

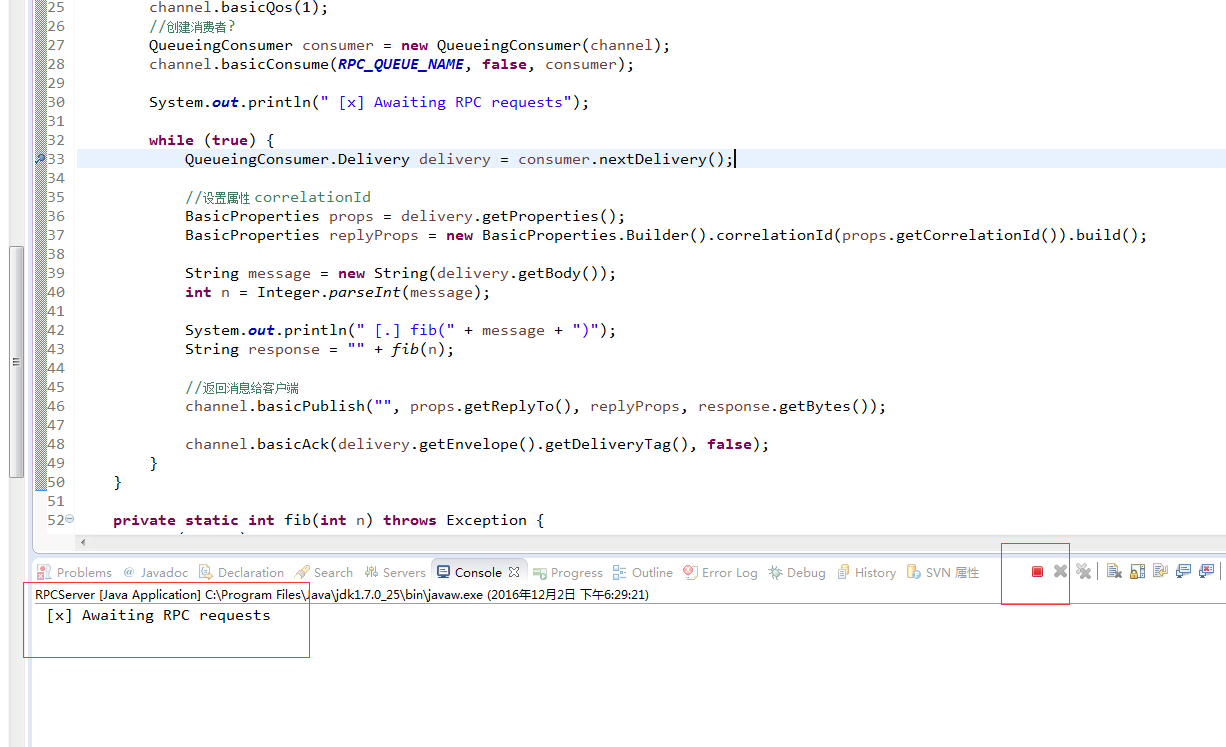
fibonacciRpc.close();

}

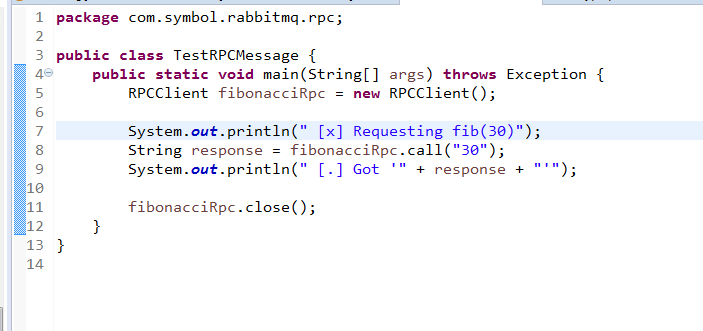
}

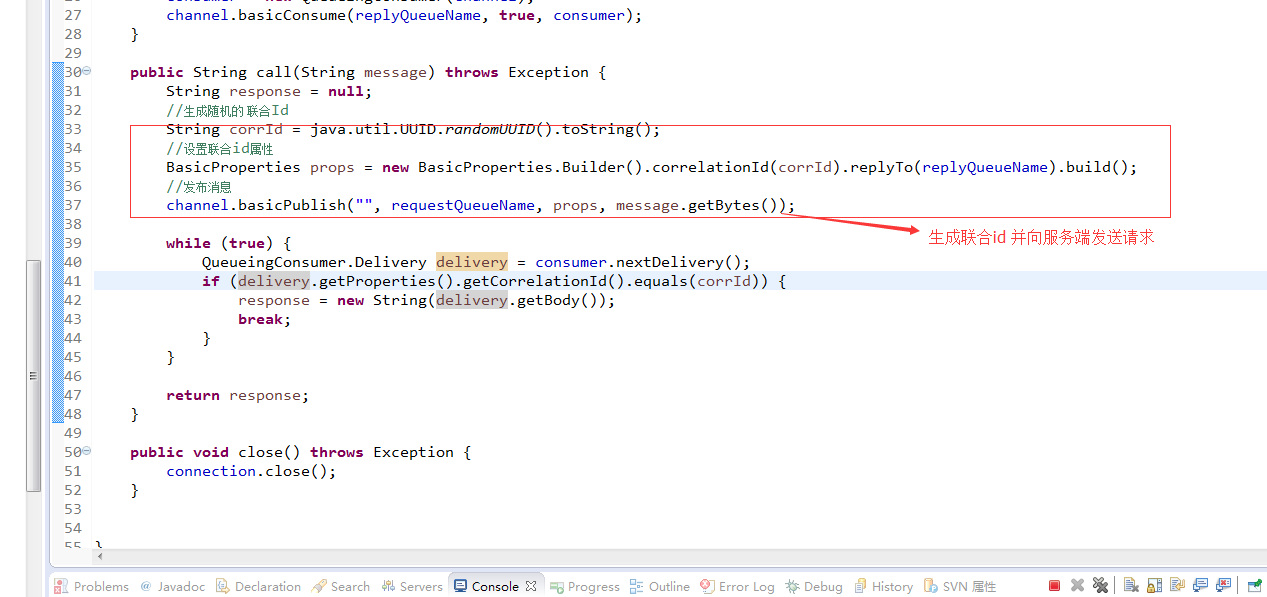
### 测试结果

启动服务端，等待请求



创建客户端实例，并 RPC



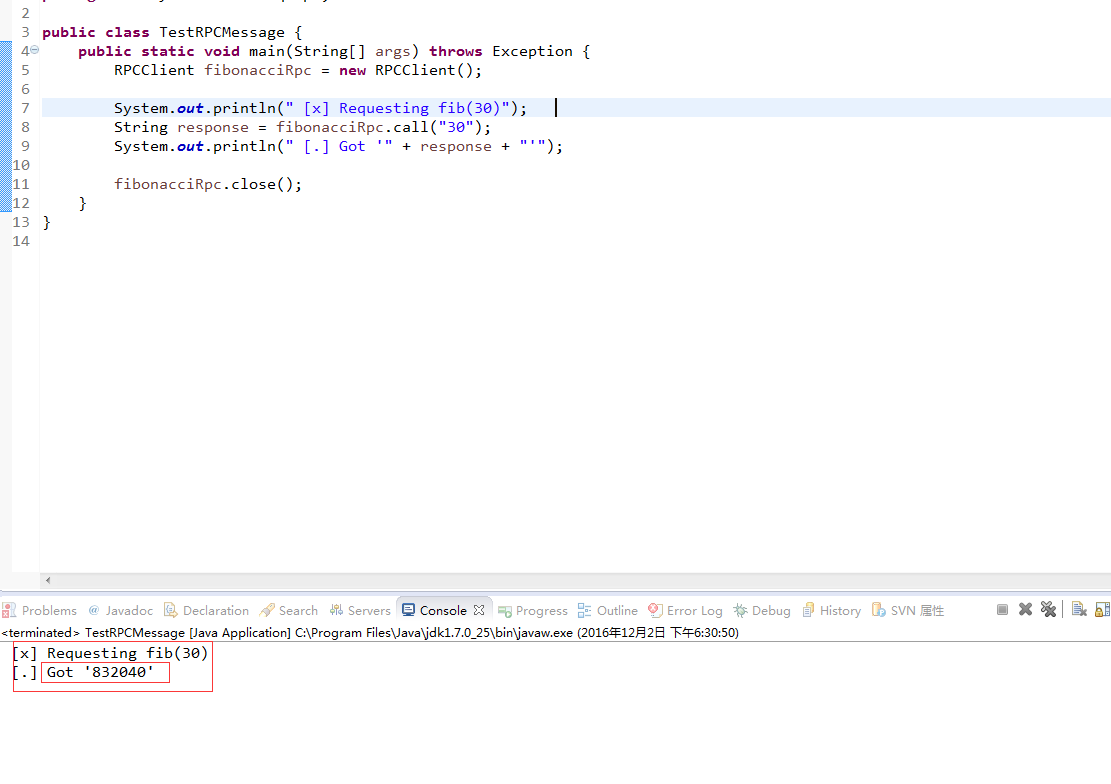


服务端接收请求



客户端答应获得的结果





6、

这里介绍的设计不是一个RPC服务的唯一可能的实现，但它有一些重要的优点

1. 如果RPC服务器太慢了，你可以通过运行另一个来扩大规模。在一个新的控制台，试着运行第二个 RPCServer
2. 在客户端，要求发送和接收的只有一个消息，非同步调用，像queueDeclare 是必要的，因此 RPC 客户端需要一个网络往返一个单一的 RPC 请求

我们的代码仍然是非常简单的，并没有试图解决更复杂（但重要的）问题，如：

1. 如果服务器没有运行，客户端该如何反应
2. 客户端是否应该有对于 RPC 的 超时(timeout) 呢？
3. 如果服务端发生故障并引发异常，应该转发到客户端吗？
4. 在处理消息之前防止传入无效的消息（检查范围、类型）