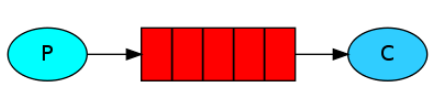
# RabbitMQ的六种模式

简单模式Simple、工作模式Work、发布订阅模式Publish/Subscribe、路由模式Routing、通配符模式Topics、远程调用模式RPC

#### **一 简单模式（Simple / HelloWorld 单生产单消费）**

简单的发送与接收，没有特别的处理。



**RabbitMQ连接（公共的连接方法，其他模式共用此方法）**

public class ConnectionUtil {

public static Connection getConnection() throws IOException, TimeoutException {

//定义连接池

ConnectionFactory connectionFactory = new ConnectionFactory();

//连接地址

connectionFactory.setHost("localhost");

//连接端口

connectionFactory.setPort(5672);

//用户名

connectionFactory.setUsername("han");

//密码

connectionFactory.setPassword("ilovelaohan");

//通过连接工厂获取连接

Connection connection = connectionFactory.newConnection();

//返回连接

return connection;

}

}

**生产者：**

public class Producter {

private final static String QUEUE\_NAME = "hello";

public static void main(String[] args) throws IOException, TimeoutException {

//获取连接

Connection connection = ConnectionUtil.getConnection();

//从连接中声明通道

Channel channel = connection.createChannel();

//队列申明

channel.queueDeclare(QUEUE\_NAME, false, false, false, null);

//消息内容

String message = "simple queue hello world !";

//推送发布消息

//basicPublish(String exchange, String routingKey, BasicProperties props, byte[] body)

channel.basicPublish("", QUEUE\_NAME, null, message.getBytes());

//信道关闭

channel.close();

//连接关闭

connection.close();

}

}

**消费者；**

public class Consumer {

private final static String QUEUE\_NAME = "hello\_queue";

public static void main(String[] args) throws IOException, TimeoutException{

//获取连接

Connection connection = ConnectionUtil.getConnection();

//声明信道

Channel channel = connection.createChannel();

//声明队列

channel.queueDeclare(QUEUE\_NAME, false, false, false, null);

//声明消费者

Consumer consumer = new DefaultConsumer(channel) {

@Override

public void handleDelivery(String consumerTag, Envelope envelope, BasicProperties properties, byte[] body)

throws IOException {

String message = new String(body,"UTF-8");

System.out.println("customer 消费消息："+message);

}

};

channel.basicConsume(QUEUE\_NAME, true,consumer);

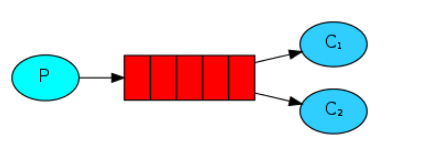
}

}

此代码建立1个生产者2个消费者的话，直接回发生轮询，1:1的概念。而且极有可能讲负载负载在某一台机器上，而另外的电脑处于闲置状态。所以必须使用ack和qos设置。

#### 二 ****工作模式（****[Work queues](https://www.rabbitmq.com/tutorials/tutorial-two-python.html)****单发送多接收）****

一个生产者端，多个消费者端。示例中为了保证消息发送的可靠性，不丢失消息，使消息持久化了。同时为了防止接收端在处理消息时down掉，只有在消息处理完成后才发送消息确认。



WorkQueueProducter

public static void main(String[] args) throws IOException, TimeoutException {

String queueName = "work\_queue";

//得到连接

Connection conn = ConnectionUtil.getConnection();

//创建频道

Channel channel = conn.createChannel();

//Queue.DeclareOk queueDeclare(String queue, boolean durable, boolean exclusive, boolean autoDelete,

//Map<String, Object> arguments) throws IOException;

//声明消息队列

boolean durable = true;

channel.queueDeclare(queueName, durable, false, false, null);

//void basicPublish(String exchange, String routingKey, BasicProperties props, byte[] body) throws IOException;

//发布消息

for (int i = 0; i < 100; i++) {

String message = "iloveyou "+i;

channel.basicPublish("", queueName, MessageProperties.PERSISTENT\_TEXT\_PLAIN, message.getBytes());

}

channel.close();

conn.close();

}

WorkQueueCustomer

public static void main(String[] args) throws IOException, TimeoutException {

String queueName = "work\_queue";

// 得到连接

Connection conn = ConnectionUtil.getConnection();

// 创建频道

Channel channel = conn.createChannel();

// Queue.DeclareOk queueDeclare(String queue, boolean durable, boolean

// exclusive, boolean autoDelete,

// Map<String, Object> arguments) throws IOException;

// 声明消息队列

boolean durable = true;

channel.queueDeclare(queueName, durable, false, false, null);

//同一时刻服务器只发送1条消息给消费者（能者多劳，消费消息快的，会消费更多的消息）

//保证在接收端一个消息没有处理完时不会接收另一个消息，即消费者端发送了ack后才会接收下一个消息。

//在这种情况下生产者端会尝试把消息发送给下一个空闲的消费者。

//channel.basicQos(1);

Consumer consumer = new DefaultConsumer(channel) {

public void handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties,

byte[] body) throws IOException {

String message = new String(body,"UTF-8");

System.out.println("消费======================1消息 "+message);

//手动返回消息

channel.basicAck(envelope.getDeliveryTag(), false);

try {

Thread.sleep(100);

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

};

//定义的消费者监听队列 autoAck：true自动返回结果，false手动返回

boolean autoAck = false;

channel.basicConsume(queueName,autoAck,consumer);

//注意，此处不要管理频道和连接

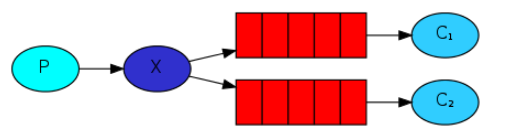
}

第二个消费者跟第一个消费者代码完全一样。所以不再赘述。

结果是：同一时刻服务器只发送1条消息给消费者（能者多劳，消费消息快的，会消费更多的消息）。要注意ack和qos的设置。

#### **三 发布、订阅模式(Publish/Subscribe)**

使用场景：发布、订阅模式，生产者端发送消息，多个消费者同时接收所有的消息



WorkQueueProducter

public static void main(String[] args) throws IOException, TimeoutException {

final String EXECHANGENAME = "publicsubscrible";

//得到连接

Connection conn = ConnectionUtil.getConnection();

//创建频道

Channel channel = conn.createChannel();

//声明交换机Exchange类型为fanout

channel.exchangeDeclare(EXECHANGENAME, "fanout");

for (int i = 0; i < 100; i++) {

String message = "i love laohan "+i;

//生产者端发布消息到交换机，使用“fanout”方式发送，

//即广播消息，不需要使用queue，发送端不需要关心谁接收。

channel.basicPublish(EXECHANGENAME, "", MessageProperties.PERSISTENT\_TEXT\_PLAIN, message.getBytes());

}

System.out.println("发布者订阅模式发布消息成功");

channel.close();

conn.close();

}

WorkQueueCustomer

public static void main(String[] args) throws IOException, TimeoutException {

final String EXECHANGENAME = "publicsubscrible";

final String queueName = "publicsubscrible\_queue";

// 得到连接

Connection conn = ConnectionUtil.getConnection();

// 创建频道

Channel channel = conn.createChannel();

//声明交换机Exchange类型为fanout

channel.exchangeDeclare(EXECHANGENAME, "fanout");

// 声明消息队列

boolean durable = true;

channel.queueDeclare(queueName, durable, false, false, null);

//绑定队列到交换机

channel.queueBind(queueName, EXECHANGENAME, "");

//注意binding queue的时候，channel.queueBind()的第三个参数Routing key为空，

//即所有的消息都接收。如果这个值不为空，在exchange type为“fanout”方式下该值被忽略！

//同一时刻服务器只发送1条消息给消费者（能者多劳，消费消息快的，会消费更多的消息）

//保证在接收端一个消息没有处理完时不会接收另一个消息，即消费者端发送了ack后才会接收下一个消息。

//在这种情况下生产者端会尝试把消息发送给下一个空闲的消费者。

channel.basicQos(1);

//申明消费者

Consumer consumer = new DefaultConsumer(channel) {

public void handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties,

byte[] body) throws IOException {

String message = new String(body,"UTF-8");

System.out.println("消费======================1消息 "+message);

//手动返回消息

channel.basicAck(envelope.getDeliveryTag(), false);

}

};

//定义的消费者监听队列 autoAck：true自动返回结果，false手动返回

boolean autoAck = false;

channel.basicConsume(queueName,autoAck,consumer);

//注意，此处不要管理频道和连接

}

第二个消费者代码跟第一个消费者完全一样，但是注意使用第二个队列。则每个消息都会被两个消费者都能获取到。

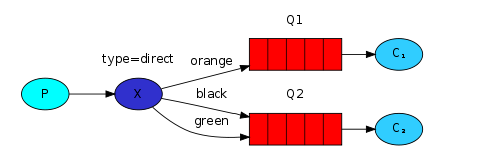
消费者端：

1、声明和生产者端一样的交换机。

2、注意binding queue的时候，channel.queueBind()的第三个参数Routing key为空，即所有的消息都接收。如果这个值不为空，在exchange type为“fanout”方式下该值被忽略！

##### **四：路由模式（Routing）**

生产者按routing key发送消息，不同的消费者端按不同的routing key接收消息。



channel.queueBind（queueName，EXCHANGE\_NAME，“”）;

绑定是交换和队列之间的关系。这可以简单地理解为：队列对来自此交换的消息感兴趣。

绑定可以采用额外的routingKey参数。为了避免与basic\_publish参数混淆，我们将其称为 绑定密钥。这就是我们如何使用键创建绑定：：

channel.queueBind（queueName，EXCHANGE\_NAME，“black”）;

绑定密钥的含义取决于交换类型。我们之前使用的 fanout 交换只是忽略了它的价值。

## 直接交换

发布者订阅模式向所有消费者广播所有消息。我们希望扩展它以允许根据消息的严重性过滤消息。例如，我们可能需要一个程序将日志消息写入磁盘以仅接收严重错误，而不是在警告或信息日志消息上浪费磁盘空间。

Fanout交换只能进行无意识的广播。

直接交换。直接交换背后的路由算法很简单 - 消息进入队列，其 绑定密钥与消息的路由密钥完全匹配。

此设置中，我们可以看到直接交换X与两个绑定到它的队列。第一个队列绑定橙色绑定，第二个绑定有两个绑定，一个绑定密钥为黑色，另一个绑定为绿色。

在这样的设置中，使用路由密钥orange发布到交换机的消息 将被路由到队列Q1。路由键为黑色 或绿色的消息将转到Q2。所有其他消息将被丢弃。

一个生产者，两个消费者：

WorkQueueProducter

final String EXECHANGENAME = "publicsubscrible4";

//得到连接

Connection conn = ConnectionUtil.getConnection();

//创建频道

Channel channel = conn.createChannel();

//声明交换机Exchange类型为direct

channel.exchangeDeclare(EXECHANGENAME, "direct");

String message = "red";

channel.basicPublish(EXECHANGENAME, "red", MessageProperties.PERSISTENT\_TEXT\_PLAIN, message.getBytes());

message = "green";

channel.basicPublish(EXECHANGENAME, "green", MessageProperties.PERSISTENT\_TEXT\_PLAIN, message.getBytes());

message = "black";

channel.basicPublish(EXECHANGENAME, "black", MessageProperties.PERSISTENT\_TEXT\_PLAIN, message.getBytes());

message = "white";

channel.basicPublish(EXECHANGENAME, "white", MessageProperties.PERSISTENT\_TEXT\_PLAIN, message.getBytes());

channel.close();

conn.close();

WorkQueueCustomer

final String EXECHANGENAME = "publicsubscrible4";

final String queueName = "publicsubscrible\_queue41";

// 得到连接

Connection conn = ConnectionUtil.getConnection();

// 创建频道

Channel channel = conn.createChannel();

//声明交换机Exchange类型为direct

channel.exchangeDeclare(EXECHANGENAME, "direct");

// 声明消息队列

boolean durable = true;

channel.queueDeclare(queueName, durable, false, false, null);

//绑定队列到交换机 指定路由routingKey

//结束路由routingKey为green和black的消息

channel.queueBind(queueName, EXECHANGENAME, "green");

channel.queueBind(queueName, EXECHANGENAME, "black");

//同一时刻服务器只发送1条消息给消费者（能者多劳，消费消息快的，会消费更多的消息）

//保证在接收端一个消息没有处理完时不会接收另一个消息，即消费者端发送了ack后才会接收下一个消息。

//在这种情况下生产者端会尝试把消息发送给下一个空闲的消费者。

channel.basicQos(1);

//申明消费者

Consumer consumer = new DefaultConsumer(channel) {

public void handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties,

byte[] body) throws IOException {

String message = new String(body,"UTF-8");

System.out.println("消费======================1消息 "+message);

//手动返回消息

channel.basicAck(envelope.getDeliveryTag(), false);

}

};

//定义的消费者监听队列 autoAck：true自动返回结果，false手动返回

boolean autoAck = false;

channel.basicConsume(queueName,autoAck,consumer);

//注意，此处不要管理频道和连接

WorkQueueCustomer2

final String EXECHANGENAME = "publicsubscrible4";

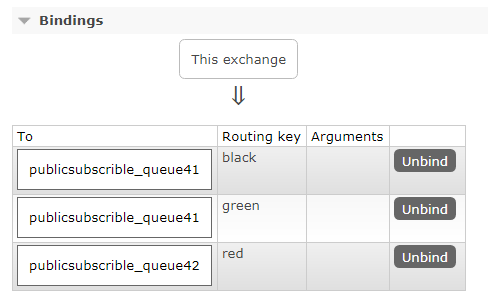
final String queueName = "publicsubscrible\_queue42";

//绑定队列到交换机 指定路由routingKey

//结束路由routingKey为green和black的消息

channel.queueBind(queueName, EXECHANGENAME, "red");

则路由模式根据routing key来判断，接受什么数据。跟消费者端和模式3（发布订阅模式）的区别：在绑定queue和exchange的时候使用了路由routing key，即从该exchange上只接收routing key指定的消息。其他没有建立队列的消息被丢弃掉。



#### 五 **通配符（或主题）模式（Topics ，按topic发送接收)**

将虚拟广播的fanout交换，更换为带路由的 direct的交换方式，可以选择性的接受消息。但它仍然具有局限性，不能基于多个标准进行路由。

通过更为复杂的 topic 交换，可以检测消息的来源。

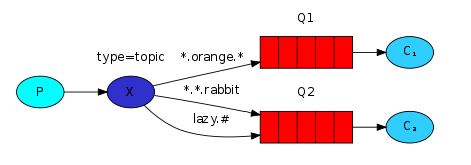
生产者端不只按固定的routing key发送消息，而是按字符串“匹配”发送，消费者端同样如此。

与之前的路由模式相比，它将信息的传输类型的key更加细化，以“key1.key2.keyN…”的模式来指定信息传输的key的大类型和大类型下面的小类型，让消费者端可以更加精细的确认自己想要获取的信息类型。而在消费者端，不用精确的指定具体到哪一个大类型下的小类型的key，而是可以使用类似正则表达式(但与正则表达式规则完全不同)的通配符在指定一定范围或符合某一个字符串匹配规则的key，来获取想要的信息。“通配符交换机”（Topic Exchange）将路由键和某模式进行匹配。此时队列需要绑定在一个模式上。符号“#”匹配一个或多个词，符号“\*”仅匹配一个词。

发送到topic交换的消息不能具有任意的 routing\_key - 它必须是由点分隔的单词列表。单词可以是任何内容，但通常它们指定与消息相关的一些功能。一些有效的路由键示例：“ stock.usd.nyse ”，“ nyse.vmw”，“ quick.orange.rabbit ”。路由密钥中可以包含任意数量的单词，最多可达255个字节。

绑定密钥也必须采用相同的形式。topic交换背后的逻辑 类似于direct交换- 使用特定路由密钥发送的消息将被传递到与匹配的绑定密钥绑定的所有队列。但是绑定键有两个重要的特殊情况

* \*（星号）可以替代一个单词。
* ＃（hash）可以替换零个或多个单词



WorkQueueProducter

final String EXECHANGENAME = "topic\_exchange";

//得到连接

Connection conn = ConnectionUtil.getConnection();

//创建频道

Channel channel = conn.createChannel();

//声明交换机Exchange类型为topic

channel.exchangeDeclare(EXECHANGENAME, "topic");

String message = "send jd order message";

channel.basicPublish(EXECHANGENAME, "order.jd", MessageProperties.PERSISTENT\_TEXT\_PLAIN, message.getBytes());

message = "send tm order message";

channel.basicPublish(EXECHANGENAME, "order.tm", MessageProperties.PERSISTENT\_TEXT\_PLAIN, message.getBytes());

message = "send jd goods message";

channel.basicPublish(EXECHANGENAME, "goods.jd", MessageProperties.PERSISTENT\_TEXT\_PLAIN, message.getBytes());

message = "send tm goods message";

channel.basicPublish(EXECHANGENAME, "goods.tm", MessageProperties.PERSISTENT\_TEXT\_PLAIN, message.getBytes());

channel.close();

conn.close();

WorkQueueCustomer

final String EXECHANGENAME = "topic\_exchange";

final String queueName = "topic\_queue1";

// 得到连接

Connection conn = ConnectionUtil.getConnection();

// 创建频道

Channel channel = conn.createChannel();

//声明交换机Exchange类型为topic

channel.exchangeDeclare(EXECHANGENAME, "topic");

// 声明消息队列

boolean durable = true;

channel.queueDeclare(queueName, durable, false, false, null);

//绑定队列到交换机 指定路由routingKey

//结束路由routingKey为green和black的消息

channel.queueBind(queueName, EXECHANGENAME, "order.\*");

//同一时刻服务器只发送1条消息给消费者（能者多劳，消费消息快的，会消费更多的消息）

//保证在接收端一个消息没有处理完时不会接收另一个消息，即消费者端发送了ack后才会接收下一个消息。

//在这种情况下生产者端会尝试把消息发送给下一个空闲的消费者。

channel.basicQos(1);

//申明消费者

Consumer consumer = new DefaultConsumer(channel) {

public void handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties,

byte[] body) throws IOException {

String message = new String(body,"UTF-8");

System.out.println("消费======================1消息 "+message);

//手动返回消息

channel.basicAck(envelope.getDeliveryTag(), false);

}

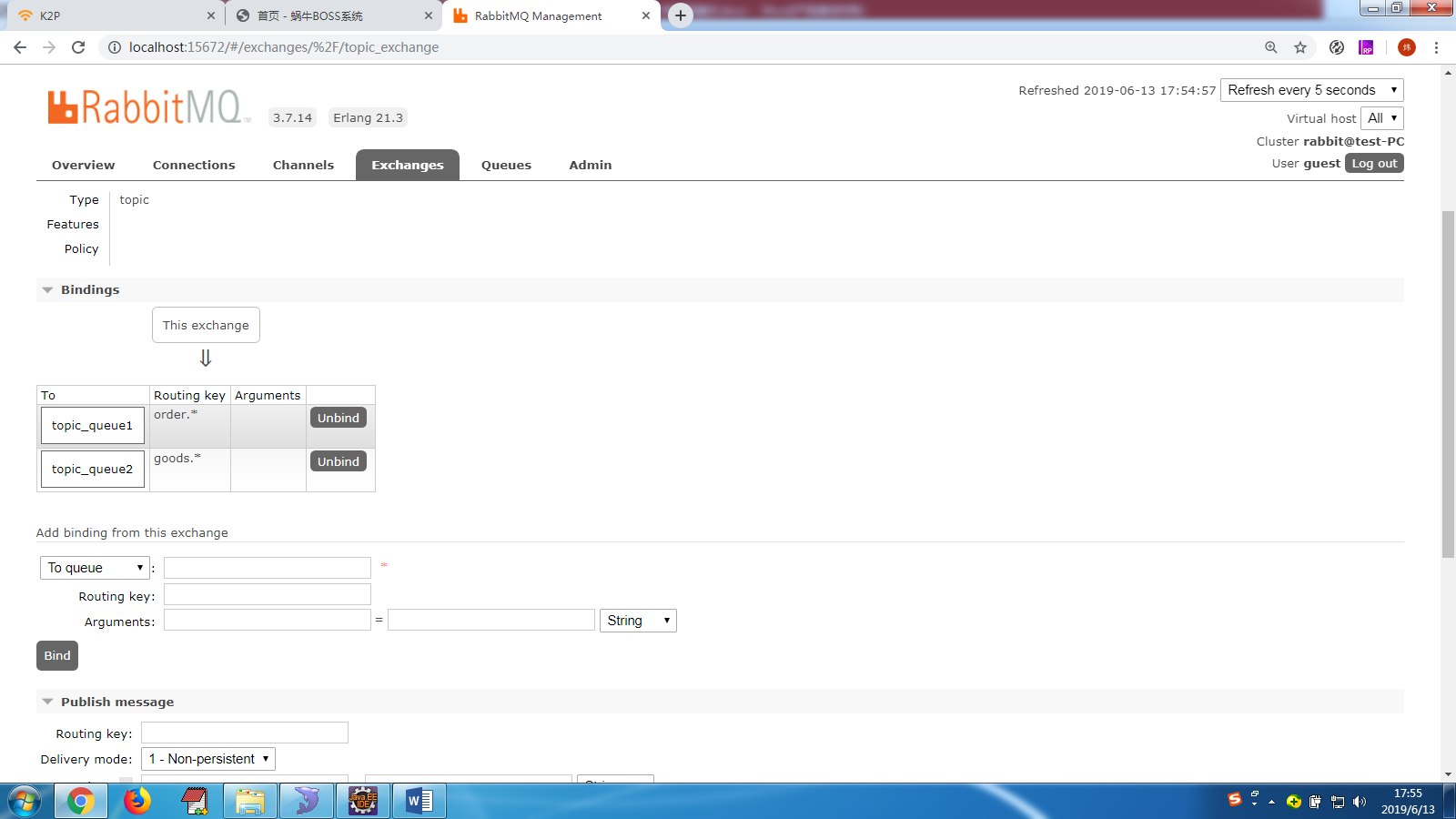
};

//定义的消费者监听队列 autoAck：true自动返回结果，false手动返回

boolean autoAck = false;

channel.basicConsume(queueName,autoAck,consumer);

//注意，此处不要管理频道和连接



生产者和模式4（路由模式）的区别：

1、交换机exchange的type为topic

2、发送消息的routing key不是固定的单词，而是匹配字符串，如"order.#"，\*匹配一个单词，#匹配0个或多个单词。因此如“order.#”能够匹配到“order.jd.buy”，但是“order.\* ”只会匹配到“order.jd”

## 六：远程过程调用（RPC）

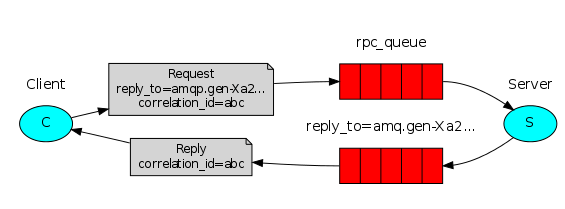
#### 有关RPC的说明

尽管RPC在计算中是一种非常常见的模式，但它经常受到批评。当程序员不知道函数调用是本地的还是慢的RPC时，会出现问题。这样的混淆导致系统不可预测，并增加了调试的不必要的复杂性。错误使用RPC可以导致不可维护的意大利面条代码，而不是简化软件。

考虑到这一点，请考虑以下建议：

* 确保明显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变得清晰。
* 处理错误案例。当RPC服务器长时间停机时，客户端应该如何反应？

如有疑问，请避免使用RPC。如果可以，您应该使用异步管道 - 而不是类似RPC的阻塞，将结果异步推送到下一个计算阶段。



我们的RPC将这样工作：

* 对于RPC请求，客户端发送带有两个属性的消息： replyTo（设置为仅为请求创建的匿名独占队列）和correlationId（设置为每个请求的唯一值）。
* 请求被发送到rpc\_queue队列。
* RPC worker（aka：server）正在等待该队列上的请求。当出现请求时，它会执行该作业，并使用来自replyTo字段的队列将带有结果的消息发送回客户端。
* 客户端等待回复队列上的数据。出现消息时，它会检查correlationId属性。如果它与请求中的值匹配，则将响应返回给应用程序。

发出客户请求：

RPCClient fibonacciRpc = new RPCClient（）;

System.out.println（“[x] Requesting fib（30）”）;

字符串响应= fibonacciRpc.call（“30”）;

System.out.println（“[。] Got'” + response + “'”）;

fibonacciRpc.close（）;