Java Client API Guide

本指南涵盖了RabbitMQ Java客户端API。 然而，这不是一个教程。 这些可在[available in a different section](https://www.rabbitmq.com/getstarted.html)找到。

该库的5.x版本系列需要JDK 8，用于编译和运行时。 在Android上，这意味着只支持 [Android 7.0](https://developer.android.com/guide/platform/j8-jack.html)或更高版本。 4.x版本系列支持Android 7.0之前的版本和JDK 6。

该库是[open-source](https://github.com/rabbitmq/rabbitmq-java-client/)，并且在三重许可(triple-licensed)下

* [Apache Public License 2.0](http://www.apache.org/licenses/LICENSE-2.0.html)
* [Mozilla Public License](https://www.mozilla.org/MPL/1.1/)
* [GPL 2.0](http://www.gnu.org/licenses/gpl-2.0.html)

这意味着用户可以考虑使用上述列表中的任何许可证进行许可。 例如，用户可以选择Apache Public License 2.0并将该客户端包含到商业产品中。 根据GPLv2许可的代码库可以选择GPLv2等。

[API reference](https://rabbitmq.github.io/rabbitmq-java-client/api/current/) (JavaDoc)可单独提供。

还有一些与Java客户端一起提供的 [command line tools](https://www.rabbitmq.com/java-tools.html)。

客户端API严格按照AMQP 0-9-1协议规范进行建模，并提供更多的abstractions以便于使用。

**[Overview](https://www.rabbitmq.com/api-guide.html" \l "classoverview)**

RabbitMQ Java客户端使用com.rabbitmq.client作为其top-level package。 关键类和接口是：

* Channel
* Connection
* ConnectionFactory
* Consumer

协议(Protocol)操作可通过Channel接口获得。 Connection用于打开通道，注册连接生命周期事件处理程序，并关闭不再需要的连接。 连接通过ConnectionFactory实例化，这就是你如何配置各种连接设置，如vhost or username。

**[Connections and Channels](https://www.rabbitmq.com/api-guide.html" \l "connecting)**

核心API类是Connection and Channel，分别代表AMQP 0-9-1 connection and channel。 它们通常在使用前被引入(imported)：

import com.rabbitmq.client.Connection;

import com.rabbitmq.client.Channel;

[**Connecting to a broker**](https://www.rabbitmq.com/api-guide.html#connecting)

以下代码使用给定参数（host name, port number,等）连接到AMQP broker：

ConnectionFactory factory = new ConnectionFactory();

factory.setUsername(userName);

factory.setPassword(password);

factory.setVirtualHost(virtualHost);

factory.setHost(hostName);

factory.setPort(portNumber);

Connection conn = factory.newConnection();

所有这些参数都对本地运行的RabbitMQ服务器具有合理的默认值。

或者，可以使用[URIs](https://www.rabbitmq.com/uri-spec.html)：

ConnectionFactory factory = new ConnectionFactory();

factory.setUri("amqp://userName:password@hostName:portNumber/virtualHost");

Connection conn = factory.newConnection();

所有这些参数都对本地运行的RabbitMQ服务器有合理的默认值。

Connection interface可以用来打开一个channel：

Channel channel = conn.createChannel();

现在可以使用该channel发送和接收消息，如后面的部分所述。

要断开连接，只需关闭channel and the connection：

channel.close();

conn.close();

请注意，关闭channel可能被认为是很好的做法，但在这里并不是必须的 - 当底层连接关闭时，它将自动完成。

**[Using Exchanges and Queues](https://www.rabbitmq.com/api-guide.html" \l "exchanges-and-queues)**

客户端应用程序可与AMQP的high-level building blocks exchanges and queues结合使用。 这些必须在可以使用之前“declared”。 声明任何一种类型的对象只是确保其中一个名称存在，并在必要时创建它。

继续前面的例子，下面的代码声明了一个exchange and a queue，然后将它们绑定(binds)在一起。

Continuing the previous example, the following code declares an exchange and a queue, then binds them together.

channel.exchangeDeclare(exchangeName, "direct", true);

String queueName = channel.queueDeclare().getQueue();

channel.queueBind(queueName, exchangeName, routingKey);

这将主动声明以下对象，这两个对象都可以通过使用其他参数进行定制。 这里他们都没有任何特别的参数(arguments)。

1. "direct" 类型的持久，non-autodelete exchange
2. 具有生成名称的non-durable exclusive, autodelete queue

上面的函数调用然后使用给定的routing key将queue绑定(bind)到exchange。

请注意，当只有一个客户端想要使用它时，这将是一种典型的声明方式：它不需要well-known name，没有其他客户端可以使用它（独占），并且会自动清除（自动删除）。 如果有几个客户想共享一个well-known name的queue，那么这个代码将是合适的：

channel.exchangeDeclare(exchangeName, "direct", true);

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

channel.queueBind(queueName, exchangeName, routingKey);

这将主动(actively 显示)声明:

1. "direct" 类型的持久，non-autodelete exchange
2. 一个具有well-known name的durable， non-exclusive, non-autodelete queue

请注意，所有这些Channel API方法都过载(overloaded)。 这些便捷的ExchangeDeclare，queueDeclare和queueBind短格式使用合理的默认值。 还有更多的参数更多的表单，可以根据需要覆盖这些默认值，在需要的地方提供完全控制。

这种"short form, long form"模式(pattern)在客户端API使用中使用。

**[Publishing messages](https://www.rabbitmq.com/api-guide.html" \l "publishing)**

要将消息发布到exchange，请按如下方式使用Channel.basicPublish：

byte[] messageBodyBytes = "Hello, world!".getBytes();

channel.basicPublish(exchangeName, routingKey, null, messageBodyBytes);

为了进行良好的控制，您可以使用重载的变体(overloaded variants)来指定强制标志(mandatory flag)，或使用pre-set message properties发送消息：

channel.basicPublish(exchangeName, routingKey, mandatory,

MessageProperties.PERSISTENT\_TEXT\_PLAIN,

messageBodyBytes);

这将发送带有delivery mode 2 (persistent)，priority 1和content-type "text/plain".的消息。 你可以使用一个Builder类来构建你自己的消息属性对象，只要你喜欢就可以提供许多属性，例如：

channel.basicPublish(exchangeName, routingKey,

new AMQP.BasicProperties.Builder()

.contentType("text/plain")

.deliveryMode(2)

.priority(1)

.userId("bob")

.build()),

messageBodyBytes);

本示例使用自定义标题(custom headers) publishes a message：

Map<String, Object> headers = new HashMap<String, Object>();

headers.put("latitude", 51.5252949);

headers.put("longitude", -0.0905493);

channel.basicPublish(exchangeName, routingKey,

new AMQP.BasicProperties.Builder()

.headers(headers)

.build()),

messageBodyBytes);

本示例publishes a message包含过期的message：

channel.basicPublish(exchangeName, routingKey,

new AMQP.BasicProperties.Builder()

.expiration("60000")

.build()),

messageBodyBytes);

我们没有在这里说明所有的可能性。

请注意，BasicProperties是autogenerated holder class AMQP的内部类。

Channel＃basicPublish的调用最终会阻止 [resource-driven alarm](http://www.rabbitmq.com/alarms.html)生效。

**[Channels and Concurrency Considerations (Thread Safety)](https://www.rabbitmq.com/api-guide.html" \l "channel-threads)**

作为一个经验法则，在线程之间共享Channel实例是需要避免的。 应用程序应该更喜欢使用每个线程的自己Channel，而不是在多个线程之间共享同一个Channel。

尽管channels上的一些操作可以安全地同时调用，但有些操作不会并且会导致不正确的帧交错(写交错)，双重确认(double acknowledgements)等等。

在shared channel上同时发布可能会导致incorrect frame interleaving on the wire，从而触connection-level protocol exception and connection closure。 因此它需要在应用程序代码中进行明确的同步（Channel＃basicPublish must be invoked in a critical section）。 在线程之间共享channels也会干扰 [Publisher Confirms](https://www.rabbitmq.com/confirms.html)。 我们强烈建议避免在共享channel上同时publishing。

在一个线程中使用并在共享channel上的另一个线程中publishing可能是安全的。

Server-pushed deliveries（请参见下面的部分）与保证per-channel ordering被preserved的保证同时进行dispatched。 调度机制使用 [java.util.concurrent.ExecutorService](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html)，每个连接一个。 可以提供一个自定义执行程序，该自定义执行程序将由使用ConnectionFactory＃setSharedExecutor设置程序的单个ConnectionFactory生成的所有连接共享。

当使用 [manual acknowledgements](https://www.rabbitmq.com/confirms.html)时，重要的是要考虑哪些线程进行确认。 如果它与received delivery的线程不同（例如Consumer＃handleDelivery委托交付处理到另一个线程），则将multiple参数设置为true进行确认是不安全的，将导致双重确认，并因此导致channel-level protocol exception 关闭channel。 一次确认一条消息可能是安全的。

**[Receiving Messages by Subscription ("Push API")](https://www.rabbitmq.com/api-guide.html" \l "consuming)**

import com.rabbitmq.client.Consumer;

import com.rabbitmq.client.DefaultConsumer;

接收消息的最有效方式是使用Consumerinterface设置订阅(subscription)。 消息将在到达时自动发送，而不是必须明确要求。

在调用与消费者有关的API方法时，个人subscriptions总是由其消费者标签(consumer tags)引用。 consumer tags是消费者标识符，可以是客户端或服务器生成的。 要让RabbitMQ生成节点范围的唯一标记，请使用Channel＃basicConsume覆盖，该覆盖不会接收使用者标记参数，也不会传递消费者标记的空字符串，并使用Channel＃basicConsume返回的值。 消费者标签用于取消消费者。

不同的消费者实例必须具有不同的消费者标签。 强烈建议在连接上重复使用消费者标签，并且可能会导致自动连接恢复问题，并在监控消费者时混淆监控数据。

实现Consumer的最简单方法是使用方便的DefaultConsumer类来创建子类。 这个子类的一个对象可以通过basicConsume调用来设置订阅(subscription)：

boolean autoAck = false;

channel.basicConsume(queueName, autoAck, "myConsumerTag",

new DefaultConsumer(channel) {

@Override

public void handleDelivery(String consumerTag,

Envelope envelope,

AMQP.BasicProperties properties,

byte[] body)

throws IOException

{

String routingKey = envelope.getRoutingKey();

String contentType = properties.getContentType();

long deliveryTag = envelope.getDeliveryTag();

*// (process the message components here ...)*

channel.basicAck(deliveryTag, false);

}

});

在这里，因为我们指定了autoAck = false，所以有必要确认传递给消费者的消息，最简单的方法是在handleDelivery方法中完成，如图所示。

更复杂的消费者将需要覆盖更多的方法。 特别是，当channels和连接关闭时调用handleShutdownSignal，并且在调用任何其他回调函数之前，handleConsumeOk会传递消费者标签(consumer tag)。

消费者也可以分别实现handleCancelOk和handleCancel方法来通知显式和隐式取消。

您可以使用Channel.basicCancel明确取消特定的消费者：

channel.basicCancel(consumerTag);

passing the consumer tag.

就像publishers一样，为消费者考虑并发危害安全也很重要。

对消费者的回调将在与实例化其Channel的线程分开的线程池中dispatched。 这意味着消费者可以安全地调用Connection或Channel上的阻塞方法(blocking methods)，例如Channel＃queueDeclare或Channel＃basicCancel。

每个Channel 都有自己的调度线程。 对于每个Channel一个消费者最常见的使用情况，这意味着消费者不支持其他消费者。 如果每个Channel有多个消费者，请注意长时间运行的消费者可能会阻止向该Channel上的其他消费者分派callbacks。

有关并发性和并发性危害安全性的其他主题，请参阅并发注意事项（线程安全性）部分。

**[Retrieving Individual Messages ("Pull API")](https://www.rabbitmq.com/api-guide.html" \l "getting)**

要显式检索消息，请使用Channel.basicGet。 返回的值是GetResponse的一个实例，从中可以提取header信息(properties)和message body：

boolean autoAck = false;

GetResponse response = channel.basicGet(queueName, autoAck);

if (response == null) {

*// No message retrieved.*

} else {

AMQP.BasicProperties props = response.getProps();

byte[] body = response.getBody();

long deliveryTag = response.getEnvelope().getDeliveryTag();

...

并且由于上面的autoAck = false，您还必须调用Channel.basicAck来确认您已成功接收消息：

...

channel.basicAck(method.deliveryTag, false); *// acknowledge receipt of the message*

}

**[Handling unroutable messages](https://www.rabbitmq.com/api-guide.html" \l "returning)**

如果message published时设置了"mandatory" flags，但无法路由(routed)，broker会将其返回给发送客户端（通过AMQP.Basic.Return命令）。

要得到这样的返回通知，客户可以实现ReturnListener接口并调用Channel.addReturnListener。 如果客户端尚未配置特定通道的返回侦听器，则相关的返回消息将被静默放弃。

channel.addReturnListener(new ReturnListener() {

public void handleReturn(int replyCode,

String replyText,

String exchange,

String routingKey,

AMQP.BasicProperties properties,

byte[] body)

throws IOException {

...

}

});

例如，如果客户端发布的消息的"mandatory" flag设置为未绑定到queue的“direct”类型的exchange，则会调用return listener。

**[Shutdown Protocol](https://www.rabbitmq.com/api-guide.html" \l "shutdown)**

**[Overview of the AMQP client shutdown](https://www.rabbitmq.com/api-guide.html" \l "shutdown-overview)**

AMQP 0-9-1 connection和channel share相同的一般方法来管理网络故障，内部故障和明确的本地关闭。

AMQP 0-9-1 connection和channel具有以下生命周期状态：

* open: 该对象已准备好使用
* closing: 该对象已明确通知贝蒂关闭，已向任何支持的下层对象发出关闭请求，并且正在等待其关闭过程完成
* closed: 该对象已经接收到来自任何较底层对象的所有关闭完成通知，并且因此关闭了它自己

这些对象总是处于关闭状态，无论导致关闭的原因如应用程序请求，内部客户端库故障，远程网络请求还是网络故障.

AMQP connection and channel objects 具有以下与shutdown-related相关的方法:

* addShutdownListener(ShutdownListener listener) and removeShutdownListener(ShutdownListener listener), 用于管理任何listeners, 当对象转换到 closed state时将会触发这些监听器.请注意，将ShutdownListener添加到已关闭的对象将立即触发监听器
* getCloseReason(), 允许调查对象关闭的愿意
* isOpen(), 用于测试对象是否处于打开状态
* close(int closeCode, String closeMessage), 显示通知要关闭的对象

listeners简单用法如下所示:

import com.rabbitmq.client.ShutdownSignalException;

import com.rabbitmq.client.ShutdownListener;

connection.addShutdownListener(new ShutdownListener() {

public void shutdownCompleted(ShutdownSignalException cause)

{

...

}

});

**[Information about the circumstances of a shutdown](https://www.rabbitmq.com/api-guide.html" \l "shutdown-cause)**

通过显式调用getCloseReason（）方法或使用ShutdownListener类的服务中的原因参数（ShutdownSignalException cause）方法，可以检索ShutdownSignalException，其中包含有关关闭原因的所有可用信息。

ShutdownSignalException类提供了分析关闭原因的方法。 通过调用isHardError（）方法，我们可以获得有关连接或通道错误的信息，getReason（）以AMQP方法的形式返回有关原因的信息 - AMQP.Channel.Close或AMQP.Connection.Close（ 如果原因是库中的某个异常（例如网络通信故障），则返回null，在这种情况下，可以使用getCause（）检索异常。

public void shutdownCompleted(ShutdownSignalException cause)

{

if (cause.isHardError())

{

Connection conn = (Connection)cause.getReference();

if (!cause.isInitiatedByApplication())

{

Method reason = cause.getReason();

...

}

...

} else {

Channel ch = (Channel)cause.getReference();

...

}

}

**[Atomicity and use of the isOpen() method](https://www.rabbitmq.com/api-guide.html" \l "shutdown-atomicity)**

不建议在生产代码中使用通道和连接对象的isOpen（）方法，因为方法返回的值取决于关闭原因的存在。 以下代码说明了竞争条件的可能性：

public void brokenMethod(Channel channel)

{

if (channel.isOpen())

{

*// The following code depends on the channel being in open state.*

*// However there is a possibility of the change in the channel state*

*// between isOpen() and basicQos(1) call*

...

channel.basicQos(1);

}

}

相反，我们通常应该忽略这种检查，并简单地尝试所需的行动。 如果在代码的执行过程中连接的通道关闭，则会引发ShutdownSignalException异常，指示对象处于无效状态。 我们还应该捕获由SocketException引发的IOException，当broker意外关闭连接时，或者在broker启动clean close时引发ShutdownSignalException。

public void validMethod(Channel channel)

{

try {

...

channel.basicQos(1);

} catch (ShutdownSignalException sse) {

*// possibly check if channel was closed*

*// by the time we started action and reasons for*

*// closing it*

...

} catch (IOException ioe) {

*// check why connection was closed*

...

}

}

**[Advanced Connection options](https://www.rabbitmq.com/api-guide.html" \l "advanced-connection)**

***[Consumer thread pool](https://www.rabbitmq.com/api-guide.html" \l "consumer-thread-pool)***

消费者线程（请参阅下面的[Receiving](https://www.rabbitmq.com/api-guide.html#consuming) ）默认情况下会自动分配到新的ExecutorServicethethread池中。 如果需要更大的控制权，请在newConnection（）方法上提供ExecutorService，以便使用此线程池。 下面是一个例子，其中提供了比通常分配的更大的线程池：

ExecutorService es = Executors.newFixedThreadPool(20);

Connection conn = factory.newConnection(es);

Executor和ExecutorService类都在java.util.concurrent包中。

当连接关闭时，默认的ExecutorService将shutdown()，但用户提供的ExecutorService（如上面的es）不会shutdown()。 提供定制ExecutorService的客户端必须确保它最终关闭（通过调用其shutdown()方法），否则池的线程可能会阻止JVM终止。

同一个executor service可以在multiple connections之间共享，或者在re-connection时被re-used，但在shutdown()后使用。

如果有证据表明processing of Consumer callbacks中存在严重瓶颈，则应仅考虑使用此功能。 如果没有 Consumer callbacks executed，或很少，默认分配绰绰有余。 开销最小，并且分配的总线程资源是有界的，即使偶尔会出现一连串的消费者活动。

**[Using Lists of Hosts](https://www.rabbitmq.com/api-guide.html" \l "address-array)**

可以将Address数组传递给newConnection（）。 地址只是带有*host* and *port* 组件的com.rabbitmq.client包中的便利类。 例如：

Address[] addrArr = new Address[]{ new Address(hostname1, portnumber1)

, new Address(hostname2, portnumber2)};

Connection conn = factory.newConnection(addrArr);

将尝试连接到 hostname1:portnumber1，并且如果无法连接到hostname2:portnumber2。 返回的连接是数组中的第一个成功（不抛出IOException）。 这完全等同于重复设置工厂的host and port，每次调用factory.newConnection（），直到其中一个成功。

如果还提供了ExecutorService（使用表单factory.newConnection（es，addrArr）），则线程池将与（第一个）成功连接相关联。

如果您想要更多地控制主机连接到，请参阅 [the support for service discovery](https://www.rabbitmq.com/api-guide.html#service-discovery-with-address-resolver)。

**[Service discovery with the AddressResolver interface](https://www.rabbitmq.com/api-guide.html" \l "service-discovery-with-address-resolver)**

从版本3.6.6开始，可以让AddressResolver的实现在创建连接时选择连接的位置：

Connection conn = factory.newConnection(addressResolver);

AddressResolver接口如下所示：

public interface AddressResolver {

List<Address> getAddresses() throws IOException;

}

就像[a list of hosts](https://www.rabbitmq.com/api-guide.html#address-array)一样，返回的第一个地址将首先尝试，然后第二个地址返回，如果客户端无法连接到第一个地址，依此类推。

如果还提供了ExecutorService（使用表单factory.newConnection（es，addressResolver）），则线程池将与（第一个）成功连接相关联。

AddressResolver是实现定制服务发现逻辑的理想场所，这对于动态基础架构尤其有用。 结合 [automatic recovery](https://www.rabbitmq.com/api-guide.html#recovery)功能，客户端可以自动连接到第一次启动时尚未达到的节点。 亲和性和负载平衡是其中可以使用自定义AddressResolver的其他场景。

Java客户端随附以下实现（有关详细信息，请参阅javadoc）：

1. DnsRecordIpAddressResolver: 给定主机的名称，返回其IP地址(针对平台DNS服务器的resolution )。这对于简单的DNS-base的负载均衡或故障转移很有用。
2. DnsSrvRecordAddressResolver: 给定服务的名称，返回hostname/port。搜索被发现为DNS SRV请求。当使用像 [HashiCorp Consul](https://www.consul.io/)这样的服务注册表是，这可能很有用。

**[Heartbeat Timeout](https://www.rabbitmq.com/api-guide.html" \l "heartbeats-timeout)**

有关检测信号以及如何在Java客户端中配置它们的更多信息，请参阅[Heartbeats guide](https://www.rabbitmq.com/heartbeats.html) 。

***[Custom Thread Factories](https://www.rabbitmq.com/api-guide.html" \l "thread-factories)***

诸如Google App Engine（GAE）等环境可以 [restrict direct thread instantiation](https://developers.google.com/appengine/docs/java/#Java_The_sandbox)。 要在这样的环境中使用RabbitMQ Java客户端，有必要配置一个自定义的ThreadFactory，它使用适当的方法来实例化线程，例如 GAE的ThreadManager。 以下是Google App Engine的一个示例。

import com.google.appengine.api.ThreadManager;

ConnectionFactory cf = new ConnectionFactory();

cf.setThreadFactory(ThreadManager.backgroundThreadFactory());

**[Support for Java non-blocking IO](https://www.rabbitmq.com/api-guide.html" \l "java-nio)**

Java客户端4.0版为Java non-blocking IO（又名Java NIO）提供了实验支持。 NIO不应该比blocking IO更快，它只是允许更容易地控制资源（在这种情况下，threads）。

在默认的blocking IO模式下，每个连接使用一个thread从network socket读取。 使用NIO模式，您可以控制从网络套接字读写数据的数量。

如果Java进程使用许多连接（数十或数百），请使用NIO模式。 您应该使用比使用默认阻止模式更少的线程。 通过设置适当的线程数量，您不应该尝试降低性能，特别是在连接不太忙时。

NIO必须明确启用：

ConnectionFactory connectionFactory = new ConnectionFactory();

connectionFactory.useNio();

NIO模式可以通过NioParams类来配置：

connectionFactory.setNioParams(new NioParams().setNbIoThreads(4));

NIO模式使用合理的默认值，但您可能需要根据您自己的工作负载进行更改。 其中一些设置是：使用的IO线程总数，缓冲区大小，用于IO循环的服务执行程序，内存写入队列的参数（写请求在网络上发送之前已排队）。 请阅读Javadoc了解详情和默认值。

**[Automatic Recovery From Network Failures](https://www.rabbitmq.com/api-guide.html" \l "recovery)**

**[Connection Recovery](https://www.rabbitmq.com/api-guide.html" \l "connection-recovery)**

客户端和RabbitMQ节点之间的网络连接可能会失败。 RabbitMQ Java客户端支持connections and topology (queues, exchanges, bindings, and consumers)的自动恢复。 许多应用程序的自动恢复过程遵循以下步骤：

1. Reconnect
2. Restore connection listeners
3. Re-open channels
4. Restore channel listeners
5. Restore channel basic.qos setting, publisher confirms and transaction settings

Topology recovery 包括为每个channel执行的以下操作

1. Re-declare exchanges (except for predefined ones)
2. Re-declare queues
3. Recover all bindings
4. Recover all consumers

*从Java客户端的4.0.0版开始，默认情况下启用自动恢复（因此也是topology recovery）。*

要disable or enable automatic connection recovery，请使用factory.setAutomaticRecoveryEnabled(boolean) method。 以下片段显示了如何明确启用自动恢复（例如，对于Java 4.0.0之前的客户端）：

ConnectionFactory factory = new ConnectionFactory();

factory.setUsername(userName);

factory.setPassword(password);

factory.setVirtualHost(virtualHost);

factory.setHost(hostName);

factory.setPort(portNumber);

factory.setAutomaticRecoveryEnabled(true);

*// connection that will recover automatically*

Connection conn = factory.newConnection();

如果因异常导致恢复失败（例如，RabbitMQ节点仍然无法访问），它将在固定时间间隔后重试（默认值为5秒）。 间隔可以配置：

ConnectionFactory factory = new ConnectionFactory();

*// attempt recovery every 10 seconds*

factory.setNetworkRecoveryInterval(10000);

当提供地址列表时，列表会被混洗，并且所有地址都会在下一个地址之后被尝试：

ConnectionFactory factory = new ConnectionFactory();

Address[] addresses = {new Address("192.168.1.4"), new Address("192.168.1.5")};

factory.newConnection(addresses);

**[When Will Connection Recovery Be Triggered?](https://www.rabbitmq.com/api-guide.html" \l "connection-recovery-triggers)**

Automatic connection recovery（如果启用）将由以下事件触发：

* An I/O exception is thrown in connection's I/O loop
* A socket read operation times out
* Missed server [heartbeats](https://www.rabbitmq.com/heartbeats.html) are detected
* Any other unexpected exception is thrown in connection's I/O loop

whichever happens first.

Channel-level exceptions不会触发任何类型的恢复，因为它们通常表示应用程序中存在语义问题（例如尝试non-existent queue中consume）。

**[Recovery Listeners](https://www.rabbitmq.com/api-guide.html" \l "recovery-listeners)**

It is possible to register one or more recovery listeners on recoverable connections and channels. When connection recovery is enabled, connections returned by ConnectionFactory#newConnection and Connection#createChannel implement com.rabbitmq.client.Recoverable, providing two methods with fairly descriptive names:

* addRecoveryListener
* removeRecoveryListener

Note that you currently need to cast connections and channels to Recoverable in order to use those methods.

**[Effects on Publishing](https://www.rabbitmq.com/api-guide.html" \l "publishers)**

Messages that are published using Channel.basicPublish when connection is down will be lost. The client does not enqueue them for delivery after connection has recovered. To ensure that published messages reach RabbitMQ applications need to use [Publisher Confirms](https://www.rabbitmq.com/confirms.html) and account for connection failures.

**[Topology Recovery](https://www.rabbitmq.com/api-guide.html" \l "topology-recovery)**

Topology recovery involves recovery of exchanges, queues, bindings and consumers. It is enabled by default when automatic recovery is enabled. *Hence topology recovery is enabled by default as of Java client 4.0.0*.

Topology recovery can be disabled explicitly if needed:

ConnectionFactory factory = new ConnectionFactory();

Connection conn = factory.newConnection();

*// enable automatic recovery (e.g. Java client prior 4.0.0)*

factory.setAutomaticRecoveryEnabled(true);

*// disable topology recovery*

factory.setTopologyRecoveryEnabled(false);

**[Failure Detection and Recovery Limitations](https://www.rabbitmq.com/api-guide.html" \l "automatic-recovery-limitations)**

Automatic connection recovery has a number of limitations and intentional design decisions that applications developers need to be aware of.

When a connection is down or lost, it [takes time to detect](https://www.rabbitmq.com/heartbeats.html). Therefore there is a window of time in which both the library and the application are unaware of effective connection failure. Any messages published during this time frame are serialised and written to the TCP socket as usual. Their delivery to the broker can only be guaranteed via [publisher confirms](https://www.rabbitmq.com/confirms.html): publishing in AMQP 0-9-1 is entirely asynchronous by design.

When a socket or I/O operation error is detected by a connection with automatic recovery enabled, recovery begins after a configurable delay, 5 seconds by default. This design assumes that even though a lot of network failures are transient and generally short lived, they do not go away in an instant. Connection recovery attempts will continue at identical time intervals until a new connection is successfully opened.

When a connection is in the recovering state, any publishes attempted on its channels will be rejected with an exception. The client currently does not perform any internal buffering of such outgoing messages. It is an application developer's responsibility to keep track of such messages and republish them when recovery succeeds. [Publisher confirms](https://www.rabbitmq.com/confirms.html) is a protocol extension that should be used by publishers that cannot afford message loss.

Connection recovery will not kick in when a channel is closed due to a channel-level exception. Such exceptions often indicate application-level issues. The library cannot make an informed decision about when that's the case.

Closed channels won't be recovered even after connection recovery kicks in. This includes both explicitly closed channels and the channel-level exception case above.

**[Manual Acknowledgements and Automatic Recovery](https://www.rabbitmq.com/api-guide.html" \l "recovery-and-acknowledgements)**

When manual acknowledgements are used, it is possible that network connection to RabbitMQ node fails between message delivery and acknowledgement. After connection recovery, RabbitMQ will reset delivery tags on all channels. This means that *basic.ack*, *basic.nack*, and *basic.reject* with old delivery tags will cause a channel exception. To avoid this, RabbitMQ Java client keeps track of and updates delivery tags to make them monotonically growing between recoveries. Channel.basicAck, Channel.basicNack, and Channel.basicReject then translate adjusted delivery tags into those used by RabbitMQ. Acknowledgements with stale delivery tags will not be sent. Applications that use manual acknowledgements and automatic recovery must be capable of handling redeliveries.

**[Channels Lifecycle and Topology Recovery](https://www.rabbitmq.com/api-guide.html" \l "recovery-channel-lifecycle)**

Automatic connection recovery is meant to be as transparent as possible for the application developer, that's why Channel instances remain the same even if several connections fail and recover behind the scenes. Technically, when automatic recovery is on, Channel instances act as proxies or decorators: they delegate the AMQP business to an actual AMQP channel implementation and implement some recovery machinery around it. That is why you shouldn't close a channel after it has created some resources (queues, exchanges, bindings) or topology recovery for those resources will fail later, as the channel has been closed. Instead, leave creating channels open for the life of the application.

**[Unhandled Exceptions](https://www.rabbitmq.com/api-guide.html" \l "unhandled-exceptions)**

Unhandled exceptions related to connection, channel, recovery, and consumer lifecycle are delegated to the exception handler. Exception handler is any object that implements the ExceptionHandler interface. By default, an instance of DefaultExceptionHandler is used. It prints exception details to the standard output.

It is possible to override the handler using ConnectionFactory#setExceptionHandler. It will be used for all connections created by the factory:

ConnectionFactory factory = new ConnectionFactory();

cf.setExceptionHandler(customHandler);

Exception handlers should be used for exception logging.

**[Metrics and monitoring](https://www.rabbitmq.com/api-guide.html" \l "metrics)**

As of version 4.0.0, the client gathers runtime metrics (e.g. number of published messages). Metrics collection is optional and is set up at the ConnectionFactory level, using the setMetricsCollector(metricsCollector) method. This method expects a MetricsCollectorinstance, which is called in several places of the client code.

The client supports [Micrometer](http://micrometer.io/) (as of version 4.3) and [Dropwizard Metrics](http://metrics.dropwizard.io/) out of the box.

Here are the collected metrics:

* Number of open connections
* Number of open channels
* Number of published messages
* Number of consumed messages
* Number of acknowledged messages
* Number of rejected messages

Both Micrometer and Dropwizard Metrics provide counts, but also mean rate, last five minute rate, etc, for messages-related metrics. They also support common tools for monitoring and reporting (JMX, Graphite, Ganglia, Datadog, etc). See the dedicated sections below for more details.

Please note the following about metrics collection:

* Don't forget to add the appropriate dependencies (in Maven, Gradle, or even as JAR files) to JVM classpath when using Micrometer or Dropwizard Metrics. Those are optional dependencies and will not be pulled automatically with the Java client. You may also need to add other dependencies depending on the reporting backend(s) used.
* Metrics collection is extensible. Implementing a custom MetricsCollector for specific needs is encouraged.
* The MetricsCollector is set at the ConnectionFactory level but can be shared across different instances.
* Metrics collection doesn't support transactions. E.g. if an acknowledgment is sent in a transaction and the transaction is then rolled back, the acknowledgment is counted in the client metrics (but not by the broker obviously). Note the acknowledgment is actually sent to the broker and then cancelled by the transaction rollback, so the client metrics are correct in term of acknowledgments sent. As a summary, don't use client metrics for critical business logic, they're not guaranteed to be perfectly accurate. They are meant to be used to simplify reasoning about a running system and make operations more efficient.

**[Micrometer support](https://www.rabbitmq.com/api-guide.html" \l "metrics-micrometer)**

You can enable metrics collection with [Micrometer](http://micrometer.io/) the following way:

ConnectionFactory connectionFactory = new ConnectionFactory();

MicrometerMetricsCollector metrics = new MicrometerMetricsCollector();

connectionFactory.setMetricsCollector(metrics);

...

metrics.getPublishedMessages(); *// get Micrometer's Counter object*

Micrometer supports [several reporting backends](http://micrometer.io/docs): Netflix Atlas, Prometheus, Datadog, Influx, JMX, etc.

You would typically pass in an instance of MeterRegistry to the MicrometerMetricsCollector. Here is an example with JMX:

JmxMeterRegistry registry = new JmxMeterRegistry();

MicrometerMetricsCollector metrics = new MicrometerMetricsCollector(registry);

ConnectionFactory connectionFactory = new ConnectionFactory();

connectionFactory.setMetricsCollector(metrics);

**[Dropwizard Metrics support](https://www.rabbitmq.com/api-guide.html" \l "metrics-dropwizard-metrics)**

You can enable metrics collection with [Dropwizard](http://metrics.dropwizard.io/) the following way:

ConnectionFactory connectionFactory = new ConnectionFactory();

StandardMetricsCollector metrics = new StandardMetricsCollector();

connectionFactory.setMetricsCollector(metrics);

...

metrics.getPublishedMessages(); *// get Metrics' Meter object*

Dropwizard Metrics supports [several reporting backends](http://metrics.dropwizard.io/3.2.3/getting-started.html): console, JMX, HTTP, Graphite, Ganglia, etc.

You would typically pass in an instance of MetricsRegistry to the StandardMetricsCollector. Here is an example with JMX:

MetricRegistry registry = new MetricRegistry();

StandardMetricsCollector metrics = new StandardMetricsCollector(registry);

ConnectionFactory connectionFactory = new ConnectionFactory();

connectionFactory.setMetricsCollector(metrics);

JmxReporter reporter = JmxReporter

.forRegistry(registry)

.inDomain("com.rabbitmq.client.jmx")

.build();

reporter.start();

**[RabbitMQ Java Client on Google App Engine](https://www.rabbitmq.com/api-guide.html" \l "gae-pitfalls)**

Using RabbitMQ Java client on Google App Engine (GAE) requires using a custom thread factory that instantiates thread using GAE's ThreadManager (see above). In addition, it is necessary to set a low heartbeat interval (4-5 seconds) to avoid running into the low InputStream read timeouts on GAE:

ConnectionFactory factory = new ConnectionFactory();

cf.setRequestedHeartbeat(5);

**[Caveats and Limitations](https://www.rabbitmq.com/api-guide.html" \l "cache-pitfalls)**

To make topology recovery possible, RabbitMQ Java client maintains a cache of declared queues, exchanges, and bindings. The cache is per-connection. Certain RabbitMQ features make it impossible for clients to observe some topology changes, e.g. when a queue is deleted due to TTL. RabbitMQ Java client tries to invalidate cache entries in the most common cases:

* When queue is deleted.
* When exchange is deleted.
* When binding is deleted.
* When consumer is cancelled on an auto-deleted queue.
* When queue or exchange is unbound from an auto-deleted exchange.

However, the client cannot track these topology changes beyond a single connection. Applications that rely on auto-delete queues or exchanges, as well as queue TTL (note: not message TTL!), and use automatic connection recovery, should explicitly delete entities know to be unused or deleted, to purge client-side topology cache. This is facilitated by Channel#queueDelete, Channel#exchangeDelete, Channel#queueUnbind, and Channel#exchangeUnbindbeing idempotent in RabbitMQ 3.3.x (deleting what's not there does not result in an exception).

**[The RPC (Request/Reply) Pattern](https://www.rabbitmq.com/api-guide.html" \l "rpc)**

As a programming convenience, the Java client API offers a class RpcClient which uses a temporary reply queue to provide simple [RPC-style communication](https://www.rabbitmq.com/tutorials/tutorial-six-java.html) facilities via AMQP 0-9-1.

The class doesn’t impose any particular format on the RPC arguments and return values. It simply provides a mechanism for sending a message to a given exchange with a particular routing key, and waiting for a response on a reply queue.

import com.rabbitmq.client.RpcClient;

RpcClient rpc = new RpcClient(channel, exchangeName, routingKey);

(The implementation details of how this class uses AMQP 0-9-1 are as follows: request messages are sent with the basic.correlation\_id field set to a value unique for this RpcClientinstance, and with basic.reply\_to set to the name of the reply queue.)

Once you have created an instance of this class, you can use it to send RPC requests by using any of the following methods:

byte[] primitiveCall(byte[] message);

String stringCall(String message)

Map mapCall(Map message)

Map mapCall(Object[] keyValuePairs)

The primitiveCall method transfers raw byte arrays as the request and response bodies. The method stringCall is a thin convenience wrapper around primitiveCall, treating the message bodies as String instances in the default character encoding.

The mapCall variants are a little more sophisticated: they encode a java.util.Map containing ordinary Java values into an AMQP 0-9-1 binary table representation, and decode the response in the same way. (Note that there are some restrictions on what value types can be used here - see the javadoc for details.)

All the marshalling/unmarshalling convenience methods use primitiveCall as a transport mechanism, and just provide a wrapping layer on top of it.

**[TLS Support](https://www.rabbitmq.com/api-guide.html" \l "tls)**

It's possible to encrypt the communication between the client and the broker [using TLS](https://www.rabbitmq.com/ssl.html). Client and server authentication (a.k.a. peer verification) is also supported. Here is the simplest way to use encryption with the Java client:

ConnectionFactory factory = new ConnectionFactory();

factory.setHost("localhost");

factory.setPort(5671);

factory.useSslProtocol();

Note the client doesn't enforce any server authentication (peer certificate chain verification) in the above sample as the default, "trust all certificates" TrustManager is used. This is convenient for local development but prone to man-in-the-middle attacks and therefore not recommended for production. To learn more about TLS support in RabbitMQ, see the [TLS guide](https://www.rabbitmq.com/ssl.html). If you only want to configure the Java client (especially the peer verification and trust manager parts), read [the appropriate section](https://www.rabbitmq.com/ssl.html#trust-levels) of the TLS guide.