Java Client API Guide解读

# Introduction简介

**RabbitMQ ==> A Message Broker 消息中间件**

**RabbitMQ** is a **message broker**(中间人，中间件): it accepts and forwards messages. You can think about it as a post office: when you put the mail that you want posting in a post box, you can be sure that Mr. or Ms. Mailperson will eventually deliver the mail to your recipient. In this analogy, RabbitMQ is a post box, a post office and a postman.

The major difference between RabbitMQ and the post office is that it doesn't deal with paper, instead it accepts, stores and forwards binary blobs of data ‒ messages.

RabbitMQ, and messaging in general, uses some jargon.

**Producer、Queue、Consumer**



### Producer：

**Producing** means nothing more than **sending**. A program that sends messages is a producer :

### **Queue**：

A queue is the name for a post box which lives inside RabbitMQ. Although messages flow through RabbitMQ and your applications, they can only be stored inside a queue. A queue is only bound by the host's memory & disk limits, **it's essentially a large message buffer**. Many producers can send messages that go to one queue, and many consumers can try to receive data from one queue. This is how we represent a queue:

### **Consumer**：

Consuming has a similar meaning to receiving. A consumer is a program that mostly waits to receive messages:

**Note** that the **producer, consumer, and broker** do not have to **reside on(居住在)** the same host; indeed in most applications they don't.

# Java Client API Guide

<http://www.rabbitmq.com/api-guide.html>

This guide covers **RabbitMQ Java client** and **its public API**. It assumes that the most recent major version of the client is used and the reader is familiar with the basics. Key sections of the guide are:

## Connecting to RabbitMQ and DisConnection from RabbitMQ

## Connection and Channel Lifespan(生命周期)

## Using Exchanges and Queues

## Publishing Messages

## Consuming Using a Subscription

## Concurrency Considerations and Safety

## Automatic Recovery From Network Failures

1. **连接RabbitMQ：Connecting to RabbitMQ**
2. **Connections and Channels**

The core API classes are **Connection** and **Channel**, representing an AMQP 0-9-1 connection and channel, respectively. They are typically imported before used:

import com.rabbitmq.client.Connection;

import com.rabbitmq.client.Channel;

1. **Connecting to RabbitMQ：获取Connection连接**
2. 通过setXxx设置ConnectionFactory的属性

The following code connects to a **RabbitMQ** node using the given parameters (host name, port number, etc):

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

// "guest"/"guest" by default, limited to **localhost** connections

factory.setUsername(userName);

factory.setPassword(password);

factory.setVirtualHost(virtualHost);

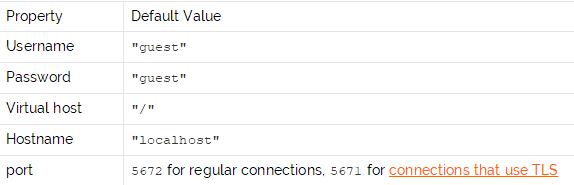
factory.setHost(hostName);

factory.setPort(portNumber);

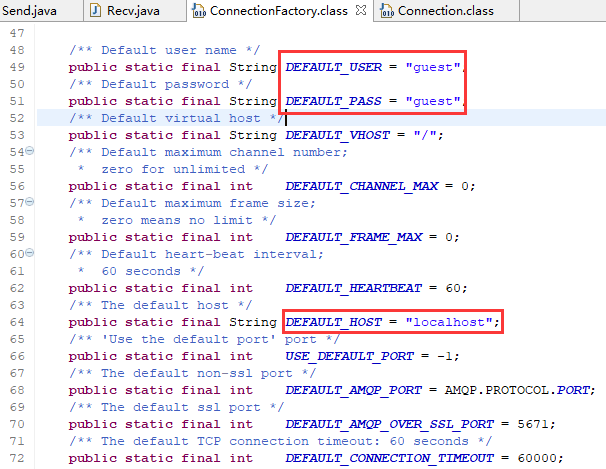
**Connection conn = factory.newConnection();**

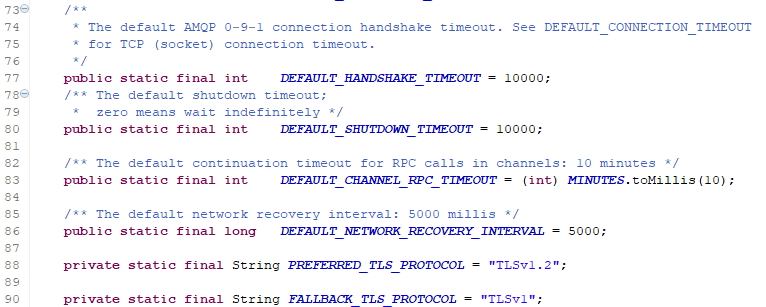
ConnectionFactory的默认参数：

All of these parameters have sensible defaults for a RabbitMQ node running locally. The default value for a property will be used if the property remains unassigned prior to creating a connection:



源代码：





1. 通过setUri方法设置ConnectionFactory属性

Alternatively, URIs may be used:

ConnectionFactory factory = new ConnectionFactory();

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

Connection conn = factory.newConnection();

All of these parameters have sensible defaults for a stock **RabbitMQ server** running locally.

Note that **user guest can only connect from localhost by default**. This is to limit well-known credential use in production systems.

1. **获取Channel**

The Connection interface can then be used to open a channel:

**Channel channel = conn.createChannel();**

The channel can now be **used to send and receive messages**, as described in subsequent sections.

Successful and unsuccessful client connection events can be observed in server node logs.

## 断开RabbitMQ：Disconnecting from RabbitMQ

To disconnect, simply close the channel and the connection:

**channel.close(); //没有严格要求关闭，因为conn关闭时channel会自动关闭**

**conn.close();**

**Note** that closing the channel may be considered good practice, but isn’t strictly necessary here - it will be done automatically anyway **when the underlying connection is closed.**

Client disconnection events can be observed **in server node logs**.

# Connection and Channel Lifespan(生命周期)

## Connection Lifespan

Connections are meant to be **long-lived.** The underlying protocol is designed and optimized for long running connections. That means that opening a new connection per operation, e.g. a message published, is unnecessary and strongly discouraged as it will introduce a lot of network roundtrips and overhead.

## Channel Lifespan

**Channels** are also meant to be **long-lived** but since many recoverable protocol errors will result in channel closure, **channel lifespan could be shorter than that of its connection**. Closing and opening new channels per operation is usually unnecessary but can be appropriate. When in doubt, consider **reusing** channels fist.

**Channel-level exceptions** such as attempts to consume from a queue that does not exist will result in channel closure. A closed channel can no longer be used and will not receive any more events from the server (such as message deliveries). **Channel-level exceptions will be logged by RabbitMQ and will initiate a shutdown sequence for the channel (see below).**

# Using Exchanges and Queues：交换机和队列的使用

## declare

**Client applications** work with **exchanges and queues**, the high-level building blocks of the protocol. These must be declared before they can be used. Declaring either type of object simply ensures that one of that name exists, creating it if necessary.

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

channel.**exchangeDeclare**(exchangeName, "**direct**", true);//Exchange Type:direct

String queueName = channel.**queueDeclare**().getQueue();//获取队列

channel.**queueBind**(queueName, exchangeName, routingKey);//将队列与Exchange绑定

This will actively declare the following objects, both of which can be customised by using additional parameters. Here neither of them have any special arguments.

### a **durable**, non-**autodelete** exchange of "direct" type

### a **non-durable**, **exclusive**, **autodelete** queue with a generated name

The above function calls then bind the queue to the exchange with the given routing key.

Note that this would be **a typical way** to declare a queue when only one client wants to work with it: it doesn’t need a well-known name, no other client can use it (exclusive) and will be cleaned up automatically (autodelete). If several clients want to share a queue with a well-known name, this code would be appropriate:

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

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

channel.**queueBind**(queueName, exchangeName, routingKey);

This will actively declare:

### a durable, non-autodelete exchange of "direct" type

### a durable, non-exclusive, non-autodelete queue with a well-known name

**Many Channel API methods** are overloaded. These convenient short forms of exchangeDeclare, queueDeclare and queueBind use sensible defaults. There are also longer forms with more parameters, to let you override these defaults as necessary, giving full control where needed.

This "**short form, long form**" pattern is used throughout the client API uses.

## Passive Declaration

Queues and exchanges can be declared "**passively**". A passive declare simply checks that the entity **with the provided name exists**. If it does, the operation is a no-op. For queues successful passive declares will return the same information as non-passive ones, namely the number of consumers and messages in ready state in the queue. If the entity does not exist, the operation fails with a channel level exception. The channel cannot be used after that. A new channel should be opened. It is common to use one-off (temporary) channels for passive declarations.

Channel#queueDeclarePassive and Channel#exchangeDeclarePassive are the methods used for passive declaration. The following example demonstrates Channel#queueDeclarePassive:

**Queue.DeclareOk response = channel.queueDeclarePassive("queue-name");**

// returns the number of messages in Ready state in the queue

**response.getMessageCount();**

// returns the number of consumers the queue has

**response.getConsumerCount();**

Channel#exchangeDeclarePassive's return value contains no useful information. Therefore if the method returns and no channel exceptions occurs, it means that the exchange does exist.

## Operations with Optional Responses

Some common operations also have a "no wait" version which won't wait for server response. For example, to declare a queue and instruct the server to not send any response, use

**channel.queueDeclareNoWait(queueName, true, false, false, null);**

The "no wait" versions are more efficient but offer lower safety guarantees, e.g. they are more dependent on the heartbeat mechanism for detection of failed operations. When in doubt, start with the standard version. The "no wait" versions are only needed in scenarios with high topology (queue, binding) churn.

## Deleting Entities and Purging Messages

A queue or exchange can be explicitly deleted:

**channel.queueDelete("queue-name")**

It is possible to delete a queue only if it is empty:

**channel.queueDelete("queue-name", false, true)**

or if it is not used (does not have any consumers):

**channel.queueDelete("queue-name", true, false)**

A queue can be purged (all of its messages deleted):(清空消息)

**channel.queuePurge("queue-name")**

purge英[pɜ:dʒ] 美[pɜ:rdʒ]vt. 肃清; 清除,净化;通便; n. 净化; 整肃; <医>泻药;

# Publishing Messages：发布消息

# Consuming Using a Subscription()

# Concurrency Considerations and Safety(并发考虑与安全)

# Automatic Recovery From Network Failures(与持久化有关)