

# RabbitMQ 中文文档

书栈(BookStack.CN)

# 目 录

[致谢](#)

[README](#)

[RabbitMQ简介 - RabbitMQ能为你做些什么？](#)

[安装](#)

[在Debian及Ubuntu系统上进行安装](#)

[RabbitMQ所支持的平台](#)

[AMQP协议](#)

[AMQP 0.9.1 模型解析](#)

[AMQP 0.9.1 快速参考指南](#)

[应用教程 - Python版](#)

[Hello World](#)

[工作队列](#)

[发布/订阅](#)

[路由](#)

[主题交换机](#)

[远程过程调用](#)

# 致谢

当前文档《RabbitMQ 中文文档》由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2018-07-09。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址：[http://www.bookstack.cn/books/RabbitMQ\\_into\\_Chinese](http://www.bookstack.cn/books/RabbitMQ_into_Chinese)

书栈官网：<http://www.bookstack.cn>

书栈开源：<https://github.com/TruthHun>

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

# README

## RabbitMQ中文

---

此项目翻译RabbitMQ中文文档及相关中文译文

### 查看文档：

---

[RabbitMQ中文 文档站](#)

### 共同参与：

---

欢迎大家共同参与RabbitMQ官方文档的翻译。（[rabbitmq.com](https://rabbitmq.com)）

- 请将翻译后的文档放置在 `translated` 目录内，并在文章末尾附上原文连接，我校对后会及时发布。
- 如果您有希望翻译成中文的文档或文章，可以在Issues中留言并附上连接。

### 项目结构

---

*"source"* 暂存格式化(markdown)后的文档

*"translated"* 暂存翻译完成但尚未校对的文档

*"published"* 存储已发布的文档

### 文档格式：

---

- 文档的文件名请使用英文
- 发布到此处的文档请统一使用 [Markdown](#) 格式。
- Markdown语法介绍可以 [点击此处](#) 查看。
- Markdown编辑器推荐：
  - 在线编辑器：[Markable](#)
  - 本地编辑器：[haroopress](#)

### 来源(书栈小编注)

---

[https://github.com/mr-ping/RabbitMQ\\_into\\_Chinese](https://github.com/mr-ping/RabbitMQ_into_Chinese)

# RabbitMQ简介 - RabbitMQ能为你做些什么？

## RabbitMQ能为你做些什么？

---

消息系统允许软件、应用相互连接和扩展。这些应用可以相互链接起来组成一个更大的应用，或者将用户设备和数据进行连接。消息系统通过将消息的发送和接收分离来实现应用程序的异步和解偶。

或许你正在考虑进行数据投递，非阻塞操作或推送通知。或许你想要实现发布 / 订阅，异步处理，或者工作队列。所有这些都可以通过消息系统实现。

RabbitMQ是一个消息代理 - 一个消息系统的媒介。它可以为你的应用提供一个通用的消息发送和接收平台，并且保证消息在传输过程中的安全。

## 技术亮点

---

### 可靠性

RabbitMQ提供了多种技术可以让你在性能和可靠性之间进行权衡。这些技术包括持久性机制、投递确认、发布者证实和高可用性机制。

### 灵活的路由

消息在到达队列前是通过交换机进行路由的。RabbitMQ为典型的路由逻辑提供了多种内置交换机类型。如果你有更复杂的路由需求，可以将这些交换机组合起来使用，你甚至可以实现自己的交换机类型，并且当做RabbitMQ的插件来使用。

### 集群

在相同局域网中的多个RabbitMQ服务器可以聚合在一起，作为一个独立的逻辑代理来使用。

### 联合

对于服务器来说，它比集群需要更多的松散和非可靠链接。为此RabbitMQ提供了联合模型。

### 高可用的队列

在同一个集群里，队列可以被镜像到多个机器中，以确保当其中某些硬件出现故障后，你的消息仍然安全。

### 多协议

RabbitMQ 支持多种消息协议的消息传递。

## 广泛的客户端

只要是你能想到的编程语言几乎都有与其相适配的RabbitMQ客户端。

## 可视化管理工具

RabbitMQ附带了一个易于使用的可视化管理工具，它可以帮助你监控消息代理的每一个环节。

## 追踪

如果你的消息系统有异常行为，RabbitMQ还提供了追踪的支持，让你能够发现问题所在。

## 插件系统

RabbitMQ附带了各种各样的插件来对自己进行扩展。你甚至也可以写自己的插件来使用。

## 还有什么呢...

---

## 商业支持

可以提供商业支持，包括培训和咨询。

## 大型社区

围绕着RabbitMQ有一个大型的社区，那儿有着各种各样的客户端、插件、指南等等。快加入我们的邮件列表参与其中吧！

## 安装

- [在Debian及Ubuntu系统上进行安装](#)
- [RabbitMQ所支持的平台](#)

# 在Debian及Ubuntu系统上进行安装

原文: [Installing on Debian / Ubuntu](#)  
状态: 校对完成  
翻译: [Ping](#)  
校对: [Ping](#)

## 安装到 Debian / Ubuntu 系统中

### 下载服务器

Description	Download	
Packaged as .deb for Debian-based Linux	<a href="#">rabbitmq-server_3.4.3-1_all.deb</a>	( <a href="#">Signature</a> )

自Debian since 6.0 (squeeze) 和 Ubuntu 9.04 之后, rabbitmq-server就已经被内置其中了。然而这些被包含在内的版本往往过低。所以从我们网站上下载 .deb 文件来安装可以达到更好的效果。查看 [Debian安装包](#) 和 [Ubuntu安装包](#) 来确认适用于指定发行版的可用版本。

你可以使用 `dpkg` 来安装从上边下载来的安装包, 也可以使用我们的APT库 (下边介绍)。

所有的依赖都会被自动安装。

### 运行RabbitMQ服务器

#### 自定义RabbitMQ环境变量

服务器会以默认方式启动。你可以[自定义RabbitMQ的环境](#)。也可以查看[如何配置组件](#)。

#### 开启服务器

当RabbitMQ安装完毕的时候服务器就会像后台程序一般运行起来了。作为一个管理员, 可以像平常一样在Debian中使用以下命令启动和关闭服务

```
invoke-rc.d rabbitmq-server stop/start/etc.
```

注意: 服务器是使用 `rabbitmq` 这个系统用户来运行的。如果你改变了Mnesia数据库或者日志的位置, 那么你必须确认这些文件属于此用户 (同时更新系统变量)。

### 我们的APT库

使用我们的APT库:

- 将以下的行添加到你的 `/etc/apt/sources.list` 文件中:

```
1. deb http://www.rabbitmq.com/debian/ testing main
```

( 请注意上边行中的 `testing` 指的是RabbitMQ发行状态, 而不是指特定的Debian发行版。你可以将它使用在



Debian的稳定版、测试版、非稳定版本中。对Ubuntu来说也是如此。我们之所以将版本描述为 `testing` 这个词是为了强调我们会频繁发布一些新的东西。)

- (可选的) 为了避免未签名的错误信息, 请使用`apt-key(8)`命令将我们的[公钥](#)添加到你的可信任密钥列表中:

```
1.  wget http://www.rabbitmq.com/rabbitmq-signing-key-public.asc
2.  sudo apt-key add rabbitmq-signing-key-public.asc
```

- 运行

```
1.  apt-get update`
```

- 像平常一样安装软件包即可; 例如

```
1.  sudo apt-get install rabbitmq-server
```

## 控制系统限制

如果要调整系统限制(尤其是打开文件的句柄的数量)的话, 可以通过编辑 `/etc/default/rabbitmq-server` 文件让服务启动的时候调用`ulimit`, 例如:

```
1.  ulimit -n 1024
```

这将会设置此服务打开文件句柄的最大数量为1024个(这也是默认设置)。

## 安全和端口

SELinux和类似机制或许会通过绑定端口的方式阻止RabbitMQ。当这种情况发生时, RabbitMQ会启动失败。请确认以下的端口是可以被打开的:

- 4369 (epmd), 25672 (Erlang distribution)
- 5672, 5671 (启用了 或者 未启用 TLS 的 AMQP 0-9-1)
- 15672 (如果管理插件被启用)
- 61613, 61614 (如果 STOMP 被启用)
- 1883, 8883 (如果 MQTT 被启用)

## 默认用户访问

代理会建立一个用户名为“guest”密码为“guest”的用户。未经配置的客户端通常会使用这个凭据。默认情况下, 这些凭据只能在链接到本机上的代理时使用, 所以在链接到其他设备的代理之前, 你需要做一些事情。

查看[访问控制](#), 了解如何新建更多的用户, 删除“guest”用户或者给“guest”用户赋予远程访问权限。

## 管理代理

如果想要停止或者查看服务器状态等，你可以调用 `rabbitmqctl`（在管理员权限下）。如果没有任何代理在运行，所有的`rabbitmqctl`命令都会给出“结点未找到”的报告。

- 调用 `rabbitmqctl stop` 来关闭服务器。
- 调用 `rabbitmqctl status` 来查看代理是否运行。

更多信息请查看 [rabbitmqctl 信息](#)

## 日志

服务器的输出被发送到 `RABBITMQ_LOG_BASE` 目录的 `RABBITMQ_NODENAME.log` 文件中。一些额外的信息会被写入到 `RABBITMQ_NODENAME-sasl.log` 文件中。

代理总是会把新的信息添加到日志文件尾部，所以完整的日志历史可以被保存下来。

你可以使用 `logrotate` 程序来执行必要的循环和压缩工作，并且你还可以更改它。默认情况下，这个位于 `/var/log/rabbitmq` 文件中的脚本会每周执行一次。你可以查看 `/etc/logrotate.d/rabbitmq-server` 来对 `logrotate` 进行配置。

# RabbitMQ所支持的平台

原文: [Supported Platforms](#)

状态: 校对完成

翻译: [Ping](#)

校对: [Ping](#)

## 支持的平台

我们的目标是让RabbitMQ运行在尽可能广泛的平台之上。RabbitMQ有着运行在所有Erlang所支持的平台之上的潜力，从嵌入式系统到多核心集群还有基于云端的服务器。

以下的平台是Erlang语言所支持的，因此RabbitMQ可以运行其上：

- Solaris
- BSD
- Linux
- MacOSX
- TRU64
- Windows NT/2000/XP/Vista/Windows 7/Windows 8
- Windows Server 2003/2008/2012
- Windows 95, 98
- VxWorks

RabbitMQ的开源版本通常被部署在以下的平台上：

- Ubuntu和其他基于Debian的Linux发行版
- Fedora和其他基于RPM包管理方式的Linux发行版
- openSUSE和衍生的发行版（包括SLES和SLERT）
- Mac OS X
- Windows XP 和 后续版本

## Windows

RabbitMQ会运行在Windows XP及其之后的版本之上（Server 2003, Vista, Windows 7, Windows 8, Server 2008 and Server 2012）。尽管没有经过测试，但它应该也可以在Windows NT 以及 Windows 2000 上良好的运行。

Windows Erlang 虚拟机能够以32位（所有可用版本）和64位（R15B往后）方式使用。将32位虚拟机运行在64位系统上的时候会有一些限制（如地址空间）存在。

## 常见的 UNIX

尽管没有官方支持，但Erlang和RabbitMQ还是可以运行在大多数系统的POSIX层上，包括Solaris, FreeBSD, NetBSD, OpenBSD等等。

## 虚拟平台

RabbitMQ可以运行在物理的或模拟的硬件中。这个特性同样允许将不支持的平台模拟成一个支持的平台来运行RabbitMQ。

如果要将RabbitMQ运行在EC2上，点击 [EC2 guide](#) 查看更多细节。

## 商业平台支持

---

[RabbitMQ commercial documentation](#)上有一系列你可以付费购买的RabbitMQ商业支持平台。

## 不支持的平台

---

一些平台是不被支持的，而且很可能永远不会：

- z/OS 和大多数的大型机
- 有内存限制的机器（小于16Mb）

如果你的平台不在此列或者你需要其他的帮助，请[联系我们](#)

# AMQP协议

- [AMQP 0.9.1 模型解析](#)
- [AMQP 0.9.1 快速参考指南](#)

# AMQP 0.9.1 模型解析

原文: [AMQP 0-9-1 Model Explained](#)

状态: 校对完成

翻译: [Ping](#)

校对: [Ping](#)

## AMQP 0-9-1 简介

关于本指南

本指南介绍了RabbitMQ所使用的 AMQP 0-9-1版本。原始版本由[Michael Klishin](#)贡献，[Chris Duncan](#)编辑。

## AMQP 0-9-1 和 AMQP 模型高阶概述

### AMQP是什么

AMQP（高级消息队列协议）是一个网络协议。它支持符合要求的客户端应用（application）和消息中间件代理（messaging middleware broker）之间进行通信。

### 消息代理和他们所扮演的角色

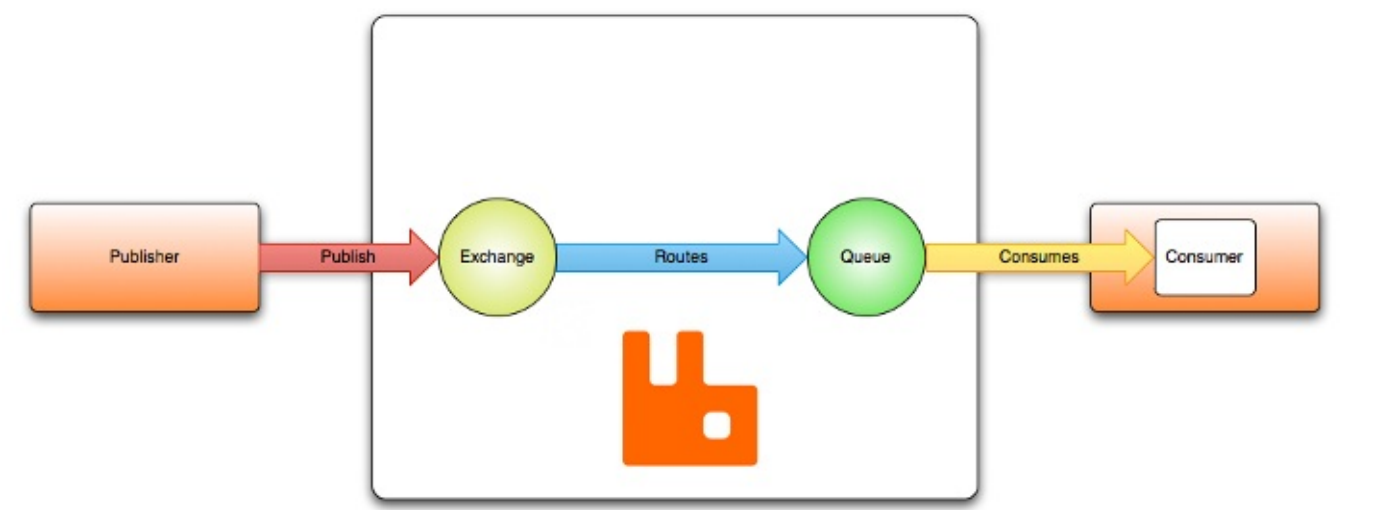
消息代理（message brokers）从发布者（publishers）亦称生产者（producers）那儿接收消息，并根据既定的路由规则把接收到的消息发送给处理消息的消费者（consumers）。

由于AMQP是一个网络协议，所以这个过程中的发布者，消费者，消息代理 可以存在于不同的设备上。

### AMQP 0-9-1 模型简介

AMQP 0-9-1的工作过程如下图：消息（message）被发布者（publisher）发送给交换机（exchange），交换机常常被比喻成邮局或者邮箱。然后交换机将收到的消息根据路由规则分发给绑定的队列（queue）。最后AMQP代理会将消息投递给订阅了此队列的消费者，或者消费者按照需求自行获取。

# "Hello, world" example routing



发布者 (publisher) 发布消息时可以给消息指定各种消息属性 (message meta-data)。有些属性有可能会被消息代理 (brokers) 使用，然而其他的属性则是完全不透明的，它们只能被接收消息的应用所使用。

从安全角度考虑，网络是不可靠的，接收消息的应用也有可能在处理消息的时候失败。基于此原因，AMQP模块包含了一个消息确认 (message acknowledgements) 的概念：当一个消息从队列中投递给消费者后 (consumer)，消费者会通知一下消息代理 (broker)，这个可以是自动的也可以由处理消息的应用的开发者执行。当“消息确认”被启用的时候，消息代理不会完全将消息从队列中删除，直到它收到来自消费者的确认回执 (acknowledgement)。

在某些情况下，例如当一个消息无法被成功路由时，消息或许会被返回给发布者并被丢弃。或者，如果消息代理执行了延期操作，消息会被放入一个所谓的死信队列中。此时，消息发布者可以选择某些参数来处理这些特殊情况。

队列，交换机和绑定统称为AMQP实体 (AMQP entities)。

## AMQP是一个可编程的协议

AMQP 0-9-1是一个可编程协议，某种意义上说AMQP的实体和路由规则是由应用本身定义的，而不是由消息代理定义。包括像声明队列和交换机，定义他们之间的绑定，订阅队列等等关于协议本身的操作。

这虽然能让开发人员自由发挥，但也需要他们注意潜在的定义冲突。当然这在实践中很少会发生，如果发生，会以配置错误 (misconfiguration) 的形式表现出来。

应用程序 (Applications) 声明AMQP实体，定义需要的路由方案，或者删除不再需要的AMQP实体。

## 交换机和交换机类型

交换机是用来发送消息的AMQP实体。交换机拿到一个消息之后将它路由给一个或零个队列。它使用哪种路由算法是由交换机类型和被称作绑定 (bindings) 的规则所决定的。AMQP 0-9-1的代理提供了四种交换机

Name ( 交换机类型 )	Default pre-declared names ( 预声明的默认名称 )
Direct exchange ( 直连交换机 )	(Empty string) and amq.direct
Fanout exchange ( 扇型交换机 )	amq.fanout

Topic exchange (主题交换机)	amq.topic
Headers exchange (头交换机)	amq.match (and amq.headers in RabbitMQ)

除交换机类型外，在声明交换机时还可以附带许多其他的属性，其中最重要的几个分别是：

- Name
- Durability (消息代理重启后，交换机是否还存在)
- Auto-delete (当所有与之绑定的消息队列都完成了对此交换机的使用后，删掉它)
- Arguments (依赖代理本身)

交换机可以有两个状态：持久 (durable)、暂存 (transient)。持久化的交换机会在消息代理 (broker) 重启后依旧存在，而暂存的交换机则不会 (它们需要在代理再次上线后重新被声明)。然而并不是所有的应用场景都需要持久化的交换机。



## 默认交换机

默认交换机 (default exchange) 实际上是一个由消息代理预先声明好的没有名字 (名字为空字符串) 的直连交换机 (direct exchange)。它有一个特殊的属性使得它对于简单应用特别有用处：那就是每个新建队列 (queue) 都会自动绑定到默认交换机上，绑定的路由键 (routing key) 名称与队列名称相同。

举个栗子：当你声明了一个名为 "search-indexing-online" 的队列，AMQP 代理会自动将其绑定到默认交换机上，绑定 (binding) 的路由键名称也是为 "search-indexing-online"。因此，当携带着名为 "search-indexing-online" 的路由键的消息被发送到默认交换机的时候，此消息会被默认交换机路由至名为 "search-indexing-online" 的队列中。换句话说，默认交换机看起来貌似能够直接将消息投递给队列，尽管技术上并没有做相关的操作。

## 直连交换机

直连型交换机 (direct exchange) 是根据消息携带的路由键 (routing key) 将消息投递给对应队列的。直连交换机用来处理消息的单播路由 (unicast routing) (尽管它也可以处理多播路由)。下边介绍它是如何工作的：

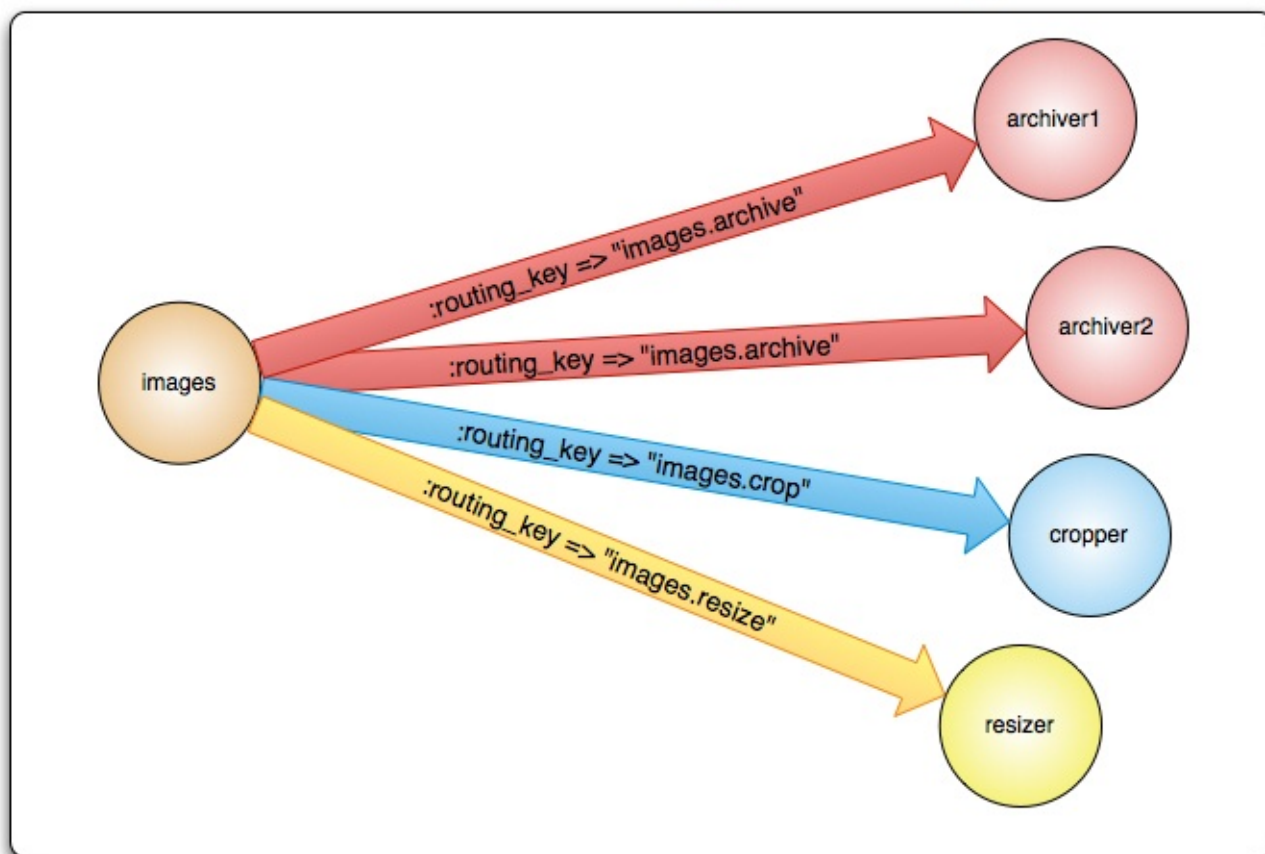
- 将一个队列绑定到某个交换机上，同时赋予该绑定一个路由键 (routing key)
- 当一个携带着路由键为  的消息被发送给直连交换机时，交换机会把它路由给绑定值同样为  的队列。

直连交换机经常用来循环分发任务给多个工作者 (workers)。当这样做的时候，我们需要明白一点，在 AMQP 0-9-1 中，消息的负载均衡是发生在消费者 (consumer) 之间的，而不是队列 (queue) 之间。

直连型交换机图例：



# Direct exchange routing



## 扇型交换机

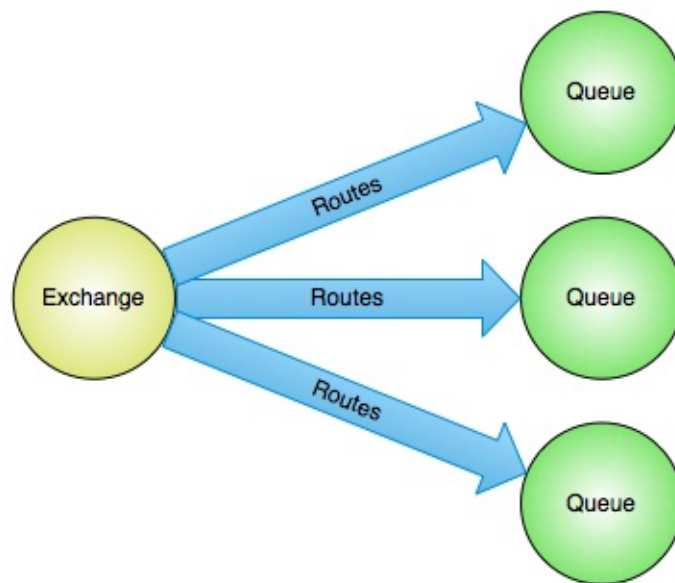
扇型交换机（fanout exchange）将消息路由给绑定到它身上的所有队列，而不理会绑定的路由键。如果N个队列绑定到某个扇型交换机上，当有消息发送给此扇型交换机时，交换机会将消息的拷贝分别发送给这所有的N个队列。扇型用来交换机处理消息的广播路由（broadcast routing）。

因为扇型交换机投递消息的拷贝到所有绑定到它的队列，所以他的应用案例都极其相似：

- 大规模多用户在线（MMO）游戏可以使用它来处理排行榜更新等全局事件
- 体育新闻网站可以用它来近乎实时地将比分更新分发给移动客户端
- 分发系统使用它来广播各种状态和配置更新
- 在群聊的时候，它被用来分发消息给参与群聊的用户。（AMQP没有内置presence的概念，因此XMPP可能会是个更好的选择）

扇型交换机图例：

# Fanout exchange routing



## 主题交换机

主题交换机 (topic exchanges) 通过对消息的路由键和队列到交换机的绑定模式之间的匹配，将消息路由给一个或多个队列。主题交换机经常用来实现各种分发/订阅模式及其变种。主题交换机通常用来实现消息的多播路由 (multicast routing)。

主题交换机拥有非常广泛的用户案例。无论何时，当一个问题涉及到那些想要有针对性的选择需要接收消息的多消费者/多应用 (multiple consumers/applications) 的时候，主题交换机都可以被列入考虑范围。

使用案例：

- 分发有关于特定地理位置的数据，例如销售点
- 由多个工作者 (workers) 完成的后台任务，每个工作者负责处理某些特定的任务
- 股票价格更新 (以及其他类型的金融数据更新)
- 涉及到分类或者标签的新闻更新 (例如，针对特定的运动项目或者队伍)
- 云端的不同种类服务的协调
- 分布式架构/基于系统的软件封装，其中每个构建者仅能处理一个特定的架构或者系统。

## 头交换机

有时消息的路由操作会涉及到多个属性，此时使用消息头就比用路由键更容易表达，头交换机 (headers exchange) 就是为此而生的。头交换机使用多个消息属性来代替路由键建立路由规则。通过判断消息头的值能否与指定的绑定相匹配来确立路由规则。

我们可以绑定一个队列到头交换机上，并给他们之间的绑定使用多个用于匹配的头 (header)。这个案例中，消息代

理得从应用开发者那儿取到更多一段信息，换句话说，它需要考虑某条消息（message）是需要部分匹配还是全部匹配。上边说的“更多一段消息”就是“x-match”参数。当“x-match”设置为“any”时，消息头的任意一个值被匹配就可以满足条件，而当“x-match”设置为“all”的时候，就需要消息头的所有值都匹配成功。

头交换机可以视为直连交换机的另一种表现形式。头交换机能够像直连交换机一样工作，不同之处在于头交换机的路由规则是建立在头属性值之上，而不是路由键。路由键必须是一个字符串，而头属性值则没有这个约束，它们甚至可以是整数或者哈希值（字典）等。

## 队列

AMQP中的队列（queue）跟其他消息队列或任务队列中的队列是很相似的：它们存储着即将被应用消费掉的消息。队列跟交换机共享某些属性，但是队列也有一些另外的属性。

- Name
- Durable（消息代理重启后，队列依旧存在）
- Exclusive（只被一个连接（connection）使用，而且当连接关闭后队列即被删除）
- Auto-delete（当最后一个消费者退订后即被删除）
- Arguments（一些消息代理用他来完成类似与TTL的某些额外功能）

队列在声明（declare）后才能被使用。如果一个队列尚不存在，声明一个队列会创建它。如果声明的队列已经存在，并且属性完全相同，那么此次声明不会对原有队列产生任何影响。如果声明中的属性与已存在队列的属性有差异，那么一个错误代码为406的通道级异常就会被抛出。

## 队列名称

队列的名字可以由应用（application）来取，也可以让消息代理（broker）直接生成一个。队列的名字可以是最多255字节的一个utf-8字符串。若希望AMQP消息代理生成队列名，需要给队列的name参数赋值一个空字符串：在同一个通道（channel）的后续的方法（method）中，我们可以使用空字符串来表示之前生成的队列名称。之所以之后的方法可以获取正确的队列名是因为通道可以默默地记住消息代理最后一次生成的队列名称。

以“amq.”开始的队列名称被预留做消息代理内部使用。如果试图在队列声明时打破这一规则的话，一个通道级的403（ACCESS\_REFUSED）错误会被抛出。

## 队列持久化

持久化队列（Durable queues）会被存储在磁盘上，当消息代理（broker）重启的时候，它依旧存在。没有被持久化的队列称作暂存队列（Transient queues）。并不是所有的场景和案例都需要将队列持久化。

持久化的队列并不会使得路由到它的消息也具有持久性。倘若消息代理挂掉了，重新启动，那么在重启的过程中持久化队列会被重新声明，无论怎样，只有经过持久化的消息才能被重新恢复。

## 绑定

绑定（Binding）是交换机（exchange）将消息（message）路由给队列（queue）所需遵循的规则。如果要指示交换机“E”将消息路由给队列“Q”，那么“Q”就需要与“E”进行绑定。绑定操作需要定义一个可选的路由键（routing key）属性给某些类型的交换机。路由键的意义在于从发送给交换机的众多消息中选择出某些消息，将其路由给绑定的

队列。

打个比方：

- 队列（queue）是我们想要去的位于纽约的目的地
- 交换机（exchange）是JFK机场
- 绑定（binding）就是JFK机场到目的地的路线。能够到达目的地的路线可以是一条或者多条

拥有了交换机这个中间层，很多由发布者直接到队列难以实现的路由方案能够得以实现，并且避免了应用开发者的许多重复劳动。

如果AMQP的消息无法路由到队列（例如，发送到的交换机没有绑定队列），消息会被就地销毁或者返还给发布者。如何处理取决于发布者设置的消息属性。

## 消费者

消息如果只是存储在队列里是没有任何用处的。被应用消费掉，消息的价值才能够体现。在AMQP 0-9-1 模型中，有两种途径可以达到此目的：

- 将消息投递给应用（“push API”）
- 应用根据需要主动获取消息（“pull API”）

使用push API，应用（application）需要明确表示出它在某个特定队列里所感兴趣的，想要消费的消息。如是，我们可以说应用注册了一个消费者，或者说订阅了一个队列。一个队列可以注册多个消费者，也可以注册一个独享的消费者（当独享消费者存在时，其他消费者即被排除在外）。

每个消费者（订阅者）都有一个叫做消费者标签的标识符。它可以被用来退订消息。消费者标签实际上是一个字符串。

## 消息确认

消费者应用（Consumer applications） - 用来接受和处理消息的应用 - 在处理消息的时候偶尔会失败或者有时会直接崩溃掉。而且网络原因也有可能引起各种问题。这就给我们出了个难题，AMQP代理在什么时候删除消息才是正确的？AMQP 0-9-1 规范给我们两种建议：

- 当消息代理（broker）将消息发送给应用后立即删除。（使用AMQP方法：`basic.deliver`或`basic.get-ok`）
- 待应用（application）发送一个确认回执（`acknowledgement`）后再删除消息。（使用AMQP方法：`basic.ack`）

前者被称作自动确认模式（`automatic acknowledgement model`），后者被称作显式确认模式（`explicit acknowledgement model`）。在显式模式下，由消费者应用来选择什么时候发送确认回执（`acknowledgement`）。应用可以在收到消息后立即发送，或将未处理的消息存储后发送，或等到消息被处理完毕后再发送确认回执（例如，成功获取一个网页内容并将其存储之后）。

如果一个消费者在尚未发送确认回执的情况下挂掉了，那AMQP代理会将消息重新投递给另一个消费者。如果当时没有可用的消费者了，消息代理会死等下一个注册到此队列的消费者，然后再次尝试投递。

## 拒绝消息

当一个消费者接收到某条消息后，处理过程有可能成功，有可能失败。应用可以向消息代理表明，本条消息由于“拒绝消息（Rejecting Messages）”的原因处理失败了（或者未能在此时完成）。当拒绝某条消息时，应用可以告诉消息代理如何处理这条消息——销毁它或者重新放入队列。当此队列只有一个消费者时，请确认不要由于拒绝消息并且选择了重新放入队列的行为而引起消息在同一个消费者身上无限循环的情况发生。

## Negative Acknowledgements

在AMQP中，`basic.reject`方法用来执行拒绝消息的操作。但`basic.reject`有个限制：你不能使用它决绝多个带有确认回执（`acknowledgements`）的消息。但是如果你使用的是RabbitMQ，那么你可以使用被称作`negative acknowledgements`（也叫`nacks`）的AMQP 0-9-1扩展来解决这个问题。更多的信息请参考[帮助页面](#)

## 预取消息

在多个消费者共享一个队列的案例中，明确指定在收到下一个确认回执前每个消费者一次可以接受多少条消息是非常有用的。这可以在试图批量发布消息的时候起到简单的负载均衡和提高消息吞吐量的作用。For example, if a producing application sends messages every minute because of the nature of the work it is doing. ( ??? 例如，如果生产应用每分钟才发送一条消息，这说明处理工作尚在运行。 )

注意，RabbitMQ只支持通道级的预取计数，而不是连接级的或者基于大小的预取。

## 消息属性和有效载荷（消息主体）

AMQP模型中的消息（`Message`）对象是带有属性（`Attributes`）的。有些属性及其常见，以至于AMQP 0-9-1 明确的定义了它们，并且应用开发者们无需费心思思考这些属性名字所代表的具体含义。例如：

- `Content type`（内容类型）
- `Content encoding`（内容编码）
- `Routing key`（路由键）
- `Delivery mode (persistent or not)`  
投递模式（持久化 或 非持久化）
- `Message priority`（消息优先权）
- `Message publishing timestamp`（消息发布的时间戳）
- `Expiration period`（消息有效期）
- `Publisher application id`（发布应用的ID）

有些属性是被AMQP代理所使用的，但是大多数是开放给接收它们的应用解释器用的。有些属性是可选的也被称作消息头（`headers`）。他们跟HTTP协议的X-Headers很相似。消息属性需要在消息被发布的时候定义。

AMQP的消息除属性外，也含有一个有效载荷 - `Payload`（消息实际携带的数据），它被AMQP代理当作不透明的字节数组来对待。消息代理不会检查或者修改有效载荷。消息可以只包含属性而不携带有效载荷。它通常会使用类似JSON这种序列化的格式数据，为了节省，协议缓冲器和`MessagePack`将结构化数据序列化，以便以消息的有效载荷的形式发布。AMQP及其同行者们通常使用“`content-type`”和“`content-encoding`”这两个字段来与消息沟通进行有效载荷的辨识工作，但这仅仅是基于约定而已。

消息能够以持久化的方式发布，AMQP代理会将此消息存储在磁盘上。如果服务器重启，系统会确认收到的持久化消息



未丢失。简单地将消息发送给一个持久化的交换机或者路由给一个持久化的队列，并不会使得此消息具有持久化性质：它完全取决于消息本身的持久模式（persistence mode）。将消息以持久化方式发布时，会对性能造成一定的影响（就像数据库操作一样，健壮性的存在必定造成一些性能牺牲）。

## 消息确认

由于网络的不确定性和应用失败的可能性，处理确认回执（acknowledgement）就变的十分重要。有时我们确认消费者收到消息就可以了，有时确认回执意味着消息已被验证并且处理完毕，例如对某些数据已经验证完毕并且进行了数据存储或者索引操作。

这种情形很常见，所以 AMQP 0-9-1 内置了一个功能叫做 消息确认（message acknowledgements），消费者用它来确认消息已经被接收或者处理。如果一个应用崩溃掉（此时连接会断掉，所以AMQP代理亦会得知），而且消息的确认回执功能已经被开启，但是消息代理尚未获得确认回执，那么消息会被从新放入队列（并且在还有有其他消费者存在于此队列的前提下，立即投递给另外一个消费者）。

协议内置的消息确认功能将帮助开发者建立强大的软件。

## AMQP 0-9-1 方法

AMQP 0-9-1由许多方法（methods）构成。方法即是操作，这跟面向对象编程中的方法没半毛钱关系。AMQP的方法被分组在类（class）中。这里的类仅仅是对AMQP方法的逻辑分组而已。在 [AMQP 0-9-1参考](#) 中有对AMQP方法的详细介绍。

让我们来看看交换机类，有一组方法被关联到了交换机的操作上。这些方法如下所示：

- exchange.declare
- exchange.declare-ok
- exchange.delete
- exchange.delete-ok

（请注意，RabbitMQ网站参考中包含了特用于RabbitMQ的交换机类的扩展，这里我们不对其进行讨论）

以上的操作来自逻辑上的配对：exchange.declare 和 exchange.declare-ok, exchange.delete 和 exchange.delete-ok. 这些操作分为“请求 - requests”（由客户端发送）和“响应 - responses”（由代理发送，用来回应之前提到的“请求”操作）。

如下的例子：客户端要求消息代理使用exchange.declare方法声明一个新的交换机：



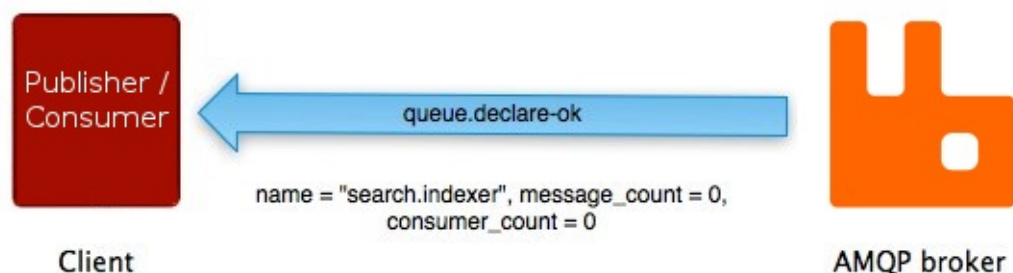
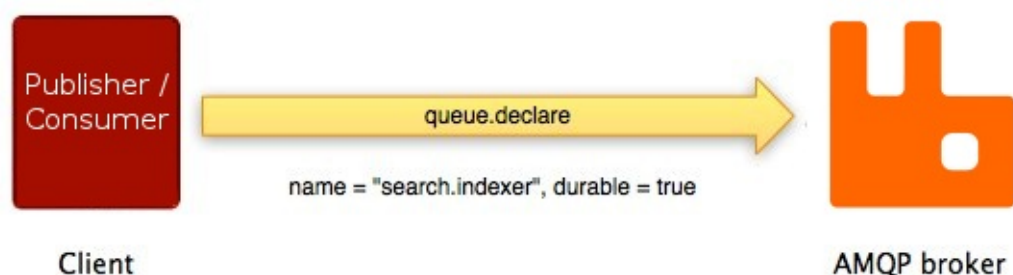
如上图所示，exchange.declare方法携带了好几个参数。这些参数可以允许客户端指定交换机名称、类型、是否持久化等等。

操作成功后，消息代理使用`exchange.declare-ok`方法进行回应：



`exchange.declare-ok`方法除了通道号之外没有携带任何其他参数（通道-`channel` 会在本指南稍后章节进行介绍）。

AMQP队列类的配对方法 - `queue.declare`方法和 `queue.declare-ok`有着与其他配对方法非常相似的一系列事件：



不是所有的AMQP方法都有与其配对的“另一半”。许多（`basic.publish`是最被广泛使用的）都没有相对应的“响应”方法，另外一些（如`basic.get`）有着一种以上与之对应的“响应”方法。

## 连接

AMQP连接通常是长连接。AMQP是一个使用TCP提供可靠投递的应用层协议。AMQP使用认证机制并且提供TLS（SSL）保护。当一个应用不再需要连接到AMQP代理的时候，需要优雅的释放掉AMQP连接，而不是直接将TCP连接关闭。

## 通道

有些应用需要与AMQP代理建立多个连接。无论如何，同时开启多个TCP连接都是不合适的，因为这样做会消耗掉过多的系统资源并且使得防火墙的配置更加困难。AMQP 0-9-1提供了通道（`channels`）来处理多连接，可以把通道理解成共享一个TCP连接的多个轻量化连接。

在涉及多线程/进程的应用中，为每个线程/进程开启一个通道（`channel`）是很常见的，并且这些通道不能被线程/

进程共享。

一个特定通道上的通讯与其他通道上的通讯是完全隔离的，因此每个AMQP方法都需要携带一个通道号，这样客户端就可以指定此方法是哪个通道准备的。

## 虚拟主机

为了在一个单独的代理上实现多个隔离的环境（用户、用户组、交换机、队列 等），AMQP提供了一个虚拟主机（virtual hosts - vhosts）的概念。这跟Web servers虚拟主机概念非常相似，这为AMQP实体提供了完全隔离的环境。当连接被建立的时候，AMQP客户端来指定使用哪个虚拟主机。

## AMQP是可扩展的

AMQP 0-9-1 拥有多个扩展点：

- 定制化交换机类型 可以让开发者们实现一些开箱即用的交换机类型尚未很好覆盖的路由方案。例如 geodata-based routing。
- 交换机和队列的声明中可以包含一些消息代理能够用到的额外属性。例如RabbitMQ中的per-queue message TTL即是使用该方式实现。
- 特定消息代理的协议扩展。例如RabbitMQ所实现的扩展。
- 新的 AMQP 0-9-1 方法类可被引入。
- 消息代理可以被其他的插件扩展，例如RabbitMQ的管理前端 和 已经被插件化的HTTP API。

这些特性使得AMQP 0-9-1模型更加灵活，并且能够适用于解决更加宽泛的问题。

## AMQP 0-9-1 客户端生态系统

AMQP 0-9-1 拥有众多的适用于各种流行语言和框架的客户端。其中一部分严格遵循AMQP规范，提供AMQP方法的实现。另一部分提供了额外的技术，方便使用的方法和抽象。有些客户端是异步的（非阻塞的），有些是同步的（阻塞的），有些将这两者同时实现。有些客户端支持“供应商的特定扩展”（例如RabbitMQ的特定扩展）。

因为AMQP的主要目标之一就是实现交互性，所以对于开发者来讲，了解协议的操作方法而不是只停留在弄懂特定客户端的库就显得十分重要。这样一来，开发者使用不同类型的库与协议进行沟通时就会容易的多。



# AMQP 0.9.1 快速参考指南

## AMQP 0-9-1 快速参考指南

本文提供了 AMQP 0-9-1 的 RabbitMQ 实现指南。作为对 [AMQP 规范](#)所定义的类和方法的有力补充，RabbitMQ 还提供了一系列[协议扩展](#)，同样在此列出。原始以及扩展规范可以在[协议页面](#)中进行下载。

为方便大家使用，相关章节内提供了 [Java](#) 和 [.Net](#) 客户端的API指南的链接。每个方法及其参数的完整细节可以从[完整的AMQP 0-9-1 参考](#)中查看。

## Basic

`basic.ack(delivery-tag delivery-tag, bit multiple)`

支持：完整

对一条或多条消息进行确认。

当确认回执(acknowledgement)由客户端发送时，此方法用来确认单条或多条消息已经由 `Deliver` 或 `Get-ok` 方法成功发送。当确认回执由服务器端发送时，此方用来确认单条或多条消息已经由 `Publish` 方法通过 `confirm` 模式下的信道(channel)成功发布。确认回执可用于单条消息甚至于一个消息集合，并且可以包含一条特定信息。

[javadoc](#) | [dotnetdoc](#) | [amqpdoc](#)

`basic.cancel(consumer-tag consumer-tag, no-wait no-wait) → cancel-ok`

支持：完整

结束队列消费者

此方法用来清除消费者。它不会影响到已经成功投递的消息，但是会使得服务器不再将新的消息投送给此消费者。客户端会在发送 `cancel` 方法和收到 `cancel-ok` 回复的过程中收到任意数量的消息。当消费者端发生不可预估的错误时，此方法也有可能由服务器发送给客户端（也就是说结束行为不是由客户端发送给服务器的`basic.cancel`方法所触发）。这种情况下客户端可以接收到由于队列被删除等原因引起的消费者丢失通知。需要注意的是，客户端从服务器接收 `basic.cancel` 方法并不是必须实现的，它通过消息代理可以辨识以协商手段接受 `basic.cancel` 的客户端的特性来正常工作。

[javadoc](#) | [dotnetdoc](#) | [amqpdoc](#)

`basic.consume(short reserved-1, queue-name queue, consumer-tag consumer-tag, no-local no-local, no-ack no-ack, bit exclusive, no-wait no-wait, table arguments) → consume-ok`

支持：部分

启动队列消费者

此方法告知服务器开启一个“消费者”，此消费者实质是一个针对特定队列消息的持久化请求。消费者在声明过的信道

(channel) 中会一直存在，直到客户端清除他们为止。

[javadoc](#) | [dotnetdoc](#) | [amqpdoc](#)

---

```
basic.deliver(consumer-tag consumer-tag, delivery-tag delivery-tag, redelivered
redelivered, exchange-name exchange, shortstr routing-key)
```

支持：完整

将消费者消息通知给客户端

此方法将一条消息通过消费者投递给客户端。在异步消息投递模式中，客户端通过 `Consume` 方法启动消费者，然后服务器使用 `Deliver` 方法将消息送达。

[amqpdoc](#)

---

```
basic.get(short reserved-1, queue-name queue, no-ack no-ack) → get-ok | get-empty
```

支持：完整

直接访问队列

此方法提供了通过同步通讯的方式直接访问队列内消息的途径。它针对的是一些有特殊需求的应用，例如对应用来说同步的功能性意义远大于应用性能。

[javadoc](#) | [dotnetdoc](#) | [amqpdoc](#)

---

```
basic.nack(delivery-tag delivery-tag, bit multiple, bit requeue)
```

此方法为RabbitMQ特有的AMQP扩展

拒绝单条或多条输入消息。

此方法允许客户端拒绝单条或多条输入消息。它可以用来打断或清除大体积消息的输入，或者将无法处理的消息返回给消息的原始队列。这个方法也可以在确认模式 (confirm mode) 下被服务器用来通知信道 (channel) 上的消息发布者有未被处理的消息存在。

[RabbitMQ Documentation](#)

[javadoc](#) | [dotnetdoc](#) | [amqpdoc](#)

---

```
basic.publish(short reserved-1, exchange-name exchange, shortstr routing-key, bit
mandatory, bit immediate)
```

支持：完整

发布单条消息

此方法用来发布单条消息到指定的交换机 (exchange)。消息将会通过配置好的交换机根据既定规则路由给队列 (queues)，之后，如果存在事务处理 (transaction)，并且事务已经被提交，就会分发给活跃的消费者。

[javadoc](#) | [dotnetdoc](#) | [amqpdoc](#)

---

`basic.qos(long prefetch-size, short prefetch-count, bit global) → qos-ok`

支持：部分  
指定服务质量

此方法指定服务的服务质量。QoS可以分配给当前的信道（channel）或者链接内的所有信道。qos方法的特定属性和语义依赖于内容类的语义。虽然qos方法原则上可以用于服务端及客户端，但在这里此方法仅适用于服务器端。

[javadoc](#) | [dotnetdoc](#) | [amqpdoc](#)

`basic.recover(bit requeue)`

支持：部分  
重新投递未被确认的消息。

此方法会要求服务器重新投递特定信道内所有未确认的消息。零条或多条消息会被重新投递。此方法用于替代异步恢复（asynchronous Recover）。

[javadoc](#) | [dotnetdoc](#) | [amqpdoc](#)

`basic.recover-async(bit requeue)`

重新投递未确认的消息。

此方法会要求服务器重新投递特定信道内所有未确认的消息。零条或多条消息会被重新投递。此方法已经弃用，取而代之的是同步 `Recover/Recover-Ok` 方法。

[javadoc](#) | [dotnetdoc](#) | [amqpdoc](#)

`basic.reject(delivery-tag delivery-tag, bit requeue)`

支持：部分  
拒绝单条输入消息。

此方法允许客户端拒绝单条或多条输入消息。它可以用来打断或清除大体积消息的输入，或者将无法处理的消息返回给消息的原始队列。

[RabbitMQ blog post](#)

[javadoc](#) | [dotnetdoc](#) | [amqpdoc](#)

`basic.return(reply-code reply-code, reply-text reply-text, exchange-name exchange, shortstr routing-key)`

支持：部分  
返回单条处理失败的消息

此方法将发布时打有 `"immediate"` 标签的无法投递的，或发布时打有 `"mandatory"` 标签的无法正确路由的单条消息返回。应答代码或文字中会注明失败原因。

## Channel

---

`channel.close(reply-code reply-code, reply-text reply-text, class-id class-id, method-id method-id) → close-ok`

支持：完整

请求关闭信道。

此方法表明发送者希望关闭信道。这通常是由于内部条件（如强制关闭）或者由于处理特定方法引起的错误（也就是 `Exception`）触发。当关闭行为是由 `exception` 触发时，发送者需提供引起 `exception` 的方法的 `class id` 和 `method id`。

[javadoc] | [dotnetdoc] | [amqpdoc]

---

`channel.flow(bit active) → flow-ok`

支持：部分

启用/禁用对端流

此方法要求对端暂停或者重启消费者发送的内容数据流。这是一个简单的流控制机制，用来避免信道的队列溢出或者发现信道接收的消息是否超出了其处理能力。需要注意的是，此方法目的不在于控制窗口。它不会影响到 `Basic.Get-ok` 方法返回的内容。

[amqpdoc]

---

`channel.open(shortstr reserved-1) → open-ok`

支持：完整

打开一个信道使用。

此方法会打开一个信道用于与服务器通讯。

[amqpdoc]

---

## Confirm

---

此类为 *RabbitMQ* 特有的 *AMQP* 扩展

`confirm.select(bit nowait) → select-ok`

.

此方法用来设置信道以便使用发布者确认回执（`acknowledgements`）。客户端仅可将此方法用于非事务性信道。

[RabbitMQ Documentation](#)

[\[javadoc\]](#) | [\[dotnetdoc\]](#) | [\[amqpdoc\]](#)

---

## Exchange

---

`exchange.bind(short reserved-1, exchange-name destination, exchange-name source, shortstr routing-key, no-wait no-wait, table arguments) → bind-ok`

此方法为*RabbitMQ*特有的*AMQP*扩展

将两个交换机进行绑定。

此方法将一个交换机绑定到另一个交换机上。

[RabbitMQ Documentation](#)

[RabbitMQ blog post](#)

[\[javadoc\]](#) | [\[dotnetdoc\]](#) | [\[amqpdoc\]](#)

---

`exchange.declare(short reserved-1, exchange-name exchange, shortstr type, bit passive, bit durable, bit auto-delete*, bit internal*, no-wait no-wait, table arguments) → declare-ok`

*RabbitMQ*针对*AMQP*协议的扩展

支持：完整

验证交换机是否存在，如有需要创建之。

如果指定交换机不存在，此方法会新建之。如果交换机已经存在，会验证其类型是否正确。

*RabbitMQ*针对*AMQP*规范实现了一个扩展，此扩展允许将无法正确路由的消息投递到一个替代交换机中（AE）。替代交换机的特性可以帮助判断客户端何时发布了无法路由的消息，并且能够提供 `"or else"` 路由语义去对某些消息做特殊处理，其他的消息则由通用方法进行处理。

[AE documention](#)

[\[javadoc\]](#) | [\[dotnetdoc\]](#) | [\[amqpdoc\]](#)

---

`exchange.delete(short reserved-1, exchange-name exchange, bit if-unused, no-wait no-wait) → delete-ok`

支持：部分

删除交换机

此方法用于删除交换机。当一个交换机被删除后，与其绑定的所有队列都会被清除。

[\[javadoc\]](#) | [\[dotnetdoc\]](#) | [\[amqpdoc\]](#)

---

`exchange.unbind(short reserved-1, exchange-name destination, exchange-name source, shortstr routing-key, no-wait no-wait, table arguments) → unbind-ok`

此方法为*RabbitMQ*特有的*AMQP*扩展

解除两个交换机之间的绑定关系

此方法用于解除两个交换机之间的绑定关系。

[javadoc] | [dotnetdoc] | [amqpdoc]

---

## Queue

---

`queue.bind(short reserved-1, queue-name queue, exchange-name exchange, shortstr routing-key, no-wait no-wait, table arguments) → bind-ok`

支持：完整

将队列绑定到交换机

此方法用于绑定队列到交换机。队列绑定到交换机之前不会接收到任何消息。在经典消息模型中，存储转发队列绑定到直连交换机，订阅队列绑定到主题交换机。

[javadoc] | [dotnetdoc] | [amqpdoc]

---

`queue.declare(short reserved-1, queue-name queue, bit passive, bit durable, bit exclusive, bit auto-delete, no-wait no-wait, table arguments) → declare-ok`

支持：完整

声明队列，如果队列不存在创建之。

此方法用于创建或检查队列。当新建一个队列时，客户端可以指定一系列属性用于控制队列的持久性及其内容，还有队列的分享等级。

RabbitMQ为AMQP规范实现了一些扩展，允许队列创建者控制队列各个方面的行为。

每个队列的消息生命周期

这个扩展决定了一条消息从发布到被服务器丢弃的生存时间。此方法中设置生存时间的参数为 `x-message-ttl` 。

队列的过期时间

队列可以在声明时指定租约时限。租约时限指的是如果队列一直未被使用，多久之后服务器会将其自动删除。租约时限由此方法的 `x-expires` 参数指定。

[x-message-ttl documentation](#)

[x-expires documentation](#)

[javadoc] | [dotnetdoc] | [amqpdoc]

---

`queue.delete(short reserved-1, queue-name queue, bit if-unused, bit if-empty, no-wait no-wait) → delete-ok`

支持：部分

删除队列。

此方法用于删除一个队列。如果服务器设置了死信队列（dead-letter queue），当队列被某个删除时，任何依存于此队列的消息都会被发送到死信队列中，队列上的所有消费者都会被清除掉。

[javadoc] | [dotnetdoc] | [amqpdoc]

---

`queue.purge(short reserved-1, queue-name queue, no-wait no-wait) → purge-ok`

支持：完整  
清空队列。

此方法会将队列中的所有不处于等待 确认回执（acknowledgment）状态的消息全部移除。

[javadoc] | [dotnetdoc] | [amqpdoc]

---

`queue.unbind(short reserved-1, queue-name queue, exchange-name exchange, shortstr routing-key, table arguments) → unbind-ok`

支持：部分  
解除队列与交换机的绑定。

此方法用于解除队列与交换机的绑定关系。

[javadoc] | [dotnetdoc] | [amqpdoc]

---

## Tx

---

`tx.commit() → commit-ok`

支持：完整  
提交当前事务。

此方法用于提交当前事务中所有的消息发布以及确（acknowledgments）认执行动作。

[javadoc] | [dotnetdoc] | [amqpdoc]

---

`tx.rollback() → rollback-ok`

支持：完整  
终止当前事务。

此方法用于终止当前事务中的所有消息发布以及确认提交操作。回滚动作完成后，一个新的事务随即开始。如果有必要，应该发布一个明确的恢复操作。

[javadoc] | [dotnetdoc] | [amqpdoc]

---

`tx.select() → select-ok`

支持：完整

选择标准事务模式。

此方法设置信道使用标准事务模式。客户端在使用提交（Commit）或者（回滚）方法之前，需要至少在信道上使用一次此方法。

[javadoc] | [dotnetdoc] | [amqpdoc]



## 应用教程 - Python版

- [Hello World](#)
- [工作队列](#)
- [发布/订阅](#)
- [路由](#)
- [主题交换机](#)
- [远程过程调用](#)

# Hello World

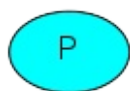
## 介绍

RabbitMQ是一个消息代理。它的工作就是接收和转发消息。你可以把它想像成一个邮局：你把信件放入邮箱，邮递员就会把信件投递到你的收件人处。在这个比喻中，RabbitMQ就扮演着邮箱、邮局以及邮递员的角色。

RabbitMQ和邮局的主要区别在于，它处理纸张，而是接收、存储和发送消息（message）这种二进制数据。

下面是RabbitMQ和消息所涉及的一些术语。

- 生产(Producing)的意思就是发送。发送消息的程序就是一个生产者(producer)。我们一般用“P”来表示：



- 队列(queue)就是存在于RabbitMQ中邮箱的名称。虽然消息的传输经过了RabbitMQ和你的应用程序，但是它只能被存储于队列当中。实质上队列就是个巨大的消息缓冲区，它的大小只受主机内存和硬盘限制。多个生产者（producers）可以把消息发送给同一个队列，同样，多个消费者（consumers）也能够从同一个队列（queue）中获取数据。队列可以绘制成这样（图上是队列的名称）：

queue\_name



- 在这里，消费（Consuming）和接收(receiving)是同一个意思。一个消费者（consumer）就是一个等待获取消息的程序。我们把它绘制为“C”：



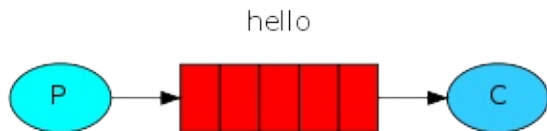
需要指出的是生产者、消费者、代理需不要待在同一个设备上；事实上大多数应用也确实不在会将他们放在一台机器上。

## Hello World!

（使用pika 0.10.0 Python客户端）

接下来我们用Python写两个小程序。一个发送单条消息的生产者（producer）和一个接收消息并将其输出的消费者（consumer）。传递的消息是“Hello World”。

下图中，“P”代表生产者，“C”代表消费者，中间的盒子代表为消费者保留的消息缓冲区，也就是我们的队列。

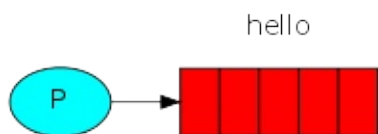


生产者（producer）把消息发送到一个名为“hello”的队列中。消费者（consumer）从这个队列中获取消息。

### RabbitMQ库

RabbitMQ使用的是AMQP 0.9.1协议。这是一个用于消息传递的开放、通用的协议。针对[不同编程语言](#)有大量的RabbitMQ客户端可用。在这个系列教程中，RabbitMQ团队推荐使用Pika这个Python客户端。大家可以通过[pip](#)这个包管理工具进行安装：

## 发送



我们第一个程序 `send.py` 会发送一个消息到队列中。首先要做的事情就是建立一个到RabbitMQ服务器的连接。

```

1. #!/usr/bin/env python
2. import pika
3.
4. connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
5. channel = connection.channel()
  
```

现在我们已经跟本地机器的代理建立了连接。如果你想连接到其他机器的代理上，需要把代表本地的 `localhost` 改为指定的名字或IP地址。

接下来，在发送消息之前，我们需要确认服务于消费者的队列已经存在。如果将消息发送给一个不存在的队列，RabbitMQ会将消息丢弃掉。下面我们创建一个名为“hello”的队列用来将消息投递进去。

```

1. channel.queue_declare(queue='hello')
  
```

这时候我们就可以发送消息了，我们第一条消息只包含了Hello World!字符串，我们打算把它发送到hello队列。

在RabbitMQ中，消息是不能直接发送到队列中的，这个过程需要通过交换机（exchange）来进行。但是为了不让细节拖累我们的进度，这里我们只需要知道如何使用由空字符串表示的默认交换机即可。如果你想要详细了解交换机，可以查看我们[教程的第三部分](#)来获取更多细节。默认交换机比较特别，它允许我们指定消息究竟需要投递到哪个具体的队列中，队列名字需要在 `routing_key` 参数中指定。

```

1. channel.basic_publish(exchange='',
2.                       routing_key='hello',
3.                       body='Hello World!')
4. print(" [x] Sent 'Hello World!'")
  
```

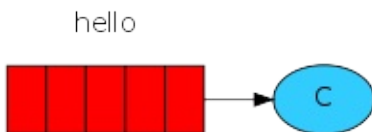
在退出程序之前，我们需要确认网络缓冲已经被刷写、消息已经投递到RabbitMQ。通过安全关闭连接可以做到这一点。

```
1. connection.close()
```

发送不成功！

如果这是你第一次使用RabbitMQ，并且没有看到“Sent”消息出现在屏幕上，你可能会抓耳挠腮不知所以。这也许是因为没有足够的磁盘空间给代理使用所造成的（代理默认需要200MB的空闲空间），所以它才会拒绝接收消息。查看一下代理的日志文件进行确认，如果需要的话也可以减少限制。[配置文件文档](#)会告诉你如何更改磁盘空间限制（`disk_free_limit`）。

## 接收



我们的第二个程序 `receive.py`，将会从队列中获取消息并将其打印到屏幕上。

这次我们还是需要要先连接到RabbitMQ服务器。连接服务器的代码和之前是一样的。

下一步也和之前一样，我们需要确认队列是存在的。我们可以多次使用 `queue_declare` 命令来创建同一个队列，但是只有一个队列会被真正的创建。

```
1. channel.queue_declare(queue='hello')
```

你也许要问：为什么要重复声明队列呢 — 我们已经在前面的代码中声明过它了。如果我们确定了队列是已经存在的，那么我们可以不这么做，比如此前预先运行了`send.py`程序。可是我们并不确定哪个程序会首先运行。这种情况下，在程序中重复将队列重复声明一下是种值得推荐的做法。

### 列出所有队列

你也许希望查看RabbitMQ中有哪些队列、有多少消息在队列中。此时你可以使用`rabbitmqctl`工具（使用有权限的用户）：

```
1. sudo rabbitmqctl list_queues
```

（在Windows中不需要`sudo`命令）

```
1. rabbitmqctl list_queues
```

从队列中获取消息相对来说稍显复杂。需要为队列定义一个回调（`callback`）函数。当我们获取到消息的时候，Pika库就会调用此回调函数。这个回调函数会将接收到的消息内容输出到屏幕上。

```
1. def callback(ch, method, properties, body):
2.     print(" [x] Received %r" % body)
```

下一步，我们需要告诉RabbitMQ这个回调函数将会从名为“hello”的队列中接收消息：

```

1. channel.basic_consume(callback,
2.                         queue='hello',
3.                         no_ack=True)

```

要成功运行这些命令，我们必须保证队列是存在的，我们的确可以确保它的存在——因为我们之前已经使用 `queue_declare` 将其声明过了。

`no_ack` 参数稍后会进行介绍。

最后，我们运行一个用来等待消息数据并且在需要的时候运行回调函数的无限循环。

```

1. print(' [*] Waiting for messages. To exit press CTRL+C')
2. channel.start_consuming()

```

## 将代码整合到一起

`send.py`的完整代码：

```

1. #!/usr/bin/env python
2. import pika
3.
4. connection =
5. pika.BlockingConnection(pika.ConnectionParameters(host='localhost'))
6. channel = connection.channel()
7.
8. channel.queue_declare(queue='hello')
9.
10. channel.basic_publish(exchange='',
11.                      routing_key='hello',
12.                      body='Hello World!')
13. print(" [x] Sent 'Hello World!'")
14. connection.close()

```

([send.py源码](#))

`receive.py`的完整代码：

```

1. #!/usr/bin/env python
2. import pika
3.
4. connection =
5. pika.BlockingConnection(pika.ConnectionParameters(host='localhost'))
6. channel = connection.channel()
7.
8. channel.queue_declare(queue='hello')
9.
10. def callback(ch, method, properties, body):
11.     print(" [x] Received %r" % body)
12.

```

```
13. channel.basic_consume(callback,
14.                         queue='hello',
15.                         no_ack=True)
16.
17. print(' [*] Waiting for messages. To exit press CTRL+C')
18. channel.start_consuming()
```

### (receive.py源码)

现在我们可以再终端中尝试一下我们的程序了。

首先我们启动一个消费者，它会持续的运行来等待投递到达。

```
1. python receive.py
2. # => [*] Waiting for messages. To exit press CTRL+C
3. # => [x] Received 'Hello World!'
```

然后启动生产者，生产者程序每次执行后都会停止运行。

```
1. python send.py
2. # => [x] Sent 'Hello World!'
```

成功了！我们已经通过RabbitMQ发送第一条消息。你也许已经注意到了，receive.py程序并没有退出。它一直在准备获取消息，你可以通过Ctrl-C来中止它。

试下在新的终端中再次运行 `send.py`。

我们已经学会如何发送消息到一个已知队列中并接收消息。是时候移步到第二部分了，我们将会建立一个简单的工作队列（work queue）。

原文: [Hello World](#)

Updated at 2017-06-16

# 工作队列

原文: [Work Queues](#)

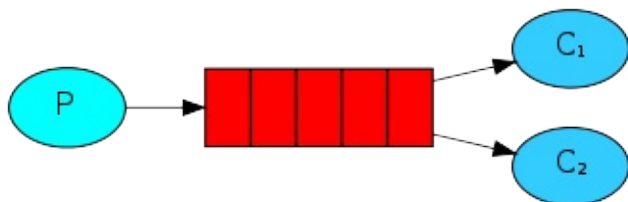
状态: 待校对

翻译: [Adam](#)

校对: [Ping](#)

## 工作队列

(使用 **pika 0.9.5** Python客户端)



在第一篇教程中，我们已经写了一个从已知队列中发送和获取消息的程序。在这篇教程中，我们将创建一个工作队列（Work Queue），它会发送一些耗时的任务给多个工作者（Worker）。

工作队列（又称：任务队列—Task Queues）是为了避免等待一些占用大量资源、时间的操作。当我们把任务（Task）当作消息发送到队列中，一个运行在后台的工作者（worker）进程就会取出任务然后处理。当你运行多个工作者（workers），任务就会在它们之间共享。

这个概念在网络应用中是非常有用的，它可以在短暂的HTTP请求中处理一些复杂的任务。

## 准备

之前的教程中，我们发送了一个包含“Hello World!”的字符串消息。现在，我们将发送一些字符串，把这些字符串当作复杂的任务。我们没有真实的例子，例如图片缩放、pdf文件转换。所以使用 `time.sleep()` 函数来模拟这种情况。我们在字符串中加上点号（.）来表示任务的复杂程度，一个点（.）将会耗时1秒钟。比如“Hello...”就会耗时3秒钟。

我们对之前教程的 `send.py` 做些简单的调整，以便可以发送随意的消息。这个程序会按照计划发送任务到我们的工作队列中。我们把它命名为 `new_task.py`：

```
1. import sys
2.
3. message = ' '.join(sys.argv[1:]) or "Hello World!"
4. channel.basic_publish(exchange='',
5.                       routing_key='hello',
6.                       body=message)
7. print " [x] Sent %r" % (message,)
```

我们的旧脚本（`receive.py`）同样需要做一些改动：它需要为消息体中每一个点号（.）模拟1秒钟的操作。它会从队列中获取消息并执行，我们把它命名为 `worker.py`：

```

1. import time
2.
3. def callback(ch, method, properties, body):
4.     print " [x] Received %r" % (body,)
5.     time.sleep( body.count('.') )
6.     print " [x] Done"

```

## 循环调度：

使用工作队列的一个好处就是它能够并行的处理队列。如果堆积了很多任务，我们只需要添加更多的工作者（workers）就可以了，扩展很简单。

首先，我们先同时运行两个worker.py脚本，它们都会从队列中获取消息，到底是不是这样呢？我们看看。

你需要打开三个终端，两个用来运行worker.py脚本，这两个终端就是我们的两个消费者（consumers）—— C1 和 C2。

```

1. shell1$ python worker.py
2. [*] Waiting for messages. To exit press CTRL+C

```

```

1. shell2$ python worker.py
2. [*] Waiting for messages. To exit press CTRL+C

```

第三个终端，我们用来发布新任务。你可以发送一些消息给消费者（consumers）：

```

1. shell3$ python new_task.py First message.
2. shell3$ python new_task.py Second message..
3. shell3$ python new_task.py Third message...
4. shell3$ python new_task.py Fourth message....
5. shell3$ python new_task.py Fifth message.....

```

看看到底发送了什么给我们的工作者（workers）：

```

1. shell1$ python worker.py
2. [*] Waiting for messages. To exit press CTRL+C
3. [x] Received 'First message.'
4. [x] Received 'Third message...'
5. [x] Received 'Fifth message.....'

```

```

1. shell2$ python worker.py
2. [*] Waiting for messages. To exit press CTRL+C
3. [x] Received 'Second message..'
4. [x] Received 'Fourth message....'

```

默认来说，RabbitMQ会按顺序得把消息发送给每个消费者（consumer）。平均每个消费者都会收到同等数量得消息。这种发送消息得方式叫做——轮询（round-robin）。试着添加三个或更多得工作者（workers）。



## 消息确认

当处理一个比较耗时得任务的时候，你也许想知道消费者（consumers）是否运行到一半就挂掉。当前的代码中，当消息被RabbitMQ发送给消费者（consumers）之后，马上就会在内存中移除。这种情况，你只要把一个工作者（worker）停止，正在处理的消息就会丢失。同时，所有发送到这个工作者的还没有处理的消息都会丢失。

我们不想丢失任何任务消息。如果一个工作者（worker）挂掉了，我们希望任务会重新发送给其他的工作者（worker）。

为了防止消息丢失，RabbitMQ提供了消息响应（acknowledgments）。消费者会通过一个ack（响应），告诉RabbitMQ已经收到并处理了某条消息，然后RabbitMQ就会释放并删除这条消息。

如果消费者（consumer）挂掉了，没有发送响应，RabbitMQ就会认为消息没有被完全处理，然后重新发送给其他消费者（consumer）。这样，及时工作者（workers）偶尔的挂掉，也不会丢失消息。

消息是没有超时这个概念的；当工作者与它断开连的时候，RabbitMQ会重新发送消息。这样在处理一个耗时非常长的消息任务的时候就不会出问题了。

消息响应默认是开启的。之前的例子中我们可以使用no\_ack=True标识把它关闭。是时候移除这个标识了，当工作者（worker）完成了任务，就发送一个响应。

```
1. def callback(ch, method, properties, body):
2.     print " [x] Received %r" % (body,)
3.     time.sleep( body.count('.') )
4.     print " [x] Done"
5.     ch.basic_ack(delivery_tag = method.delivery_tag)
6.
7. channel.basic_consume(callback,
8.                         queue='hello')
```

运行上面的代码，我们发现即使使用CTRL+C杀掉了一个工作者（worker）进程，消息也不会丢失。当工作者（worker）挂掉这后，所有没有响应的消息都会重新发送。

### 忘记确认

一个很容易犯的错误就是忘了*basic\_ack*，后果很严重。消息在你的程序退出之后就会重新发送，如果它不能够释放没响应的消息，RabbitMQ就会占用越来越多的内存。

为了排除这种错误，你可以使用*rabbitmqctl*命令，输出*messages\_unacknowledged*字段：

```
1. $ sudo rabbitmqctl list_queues name messages_ready messages_unacknowledged
2. Listing queues ...
3. hello      0      0
4. ...done.
```

## 消息持久化

如果你没有特意告诉RabbitMQ，那么在它退出或者崩溃的时候，将会丢失所有队列和消息。为了确保信息不会丢失，有两个事情是需要注意的：我们必须把“队列”和“消息”设为持久化。

首先，为了不让队列消失，需要把队列声明为持久化（durable）：

```
1. channel.queue_declare(queue='hello', durable=True)
```

尽管这行代码本身是正确的，但是仍然不会正确运行。因为我们已经定义过一个叫hello的非持久化队列。RabbitMQ不允许你使用不同的参数重新定义一个队列，它会返回一个错误。但我们现在使用一个快捷的解决方法——用不同的名字，例如task\_queue。

```
1. channel.queue_declare(queue='task_queue', durable=True)
```

这个queue\_declare必须在生产者（producer）和消费者（consumer）对应的代码中修改。

这时候，我们就可以确保在RabbitMQ重启之后queue\_declare队列不会丢失。另外，我们需要把我们的消息也要设为持久化——将delivery\_mode的属性设为2。

```
1. channel.basic_publish(exchange='',
2.                       routing_key="task_queue",
3.                       body=message,
4.                       properties=pika.BasicProperties(
5.                           delivery_mode = 2, # make message persistent
6.                       ))
```

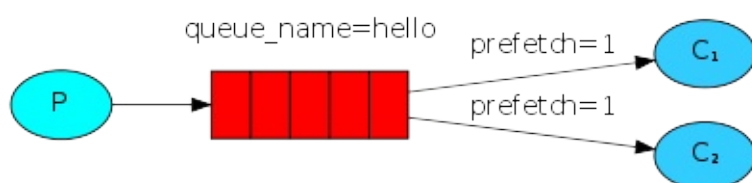
### 注意：消息持久化

将消息设为持久化并不能完全保证不会丢失。以上代码只是告诉了RabbitMQ要把消息存到硬盘，但从RabbitMQ收到消息到保存之间还是有一个很小的间隔时间。因为RabbitMQ并不是所有的消息都使用fsync(2)——它有可能只是保存到缓存中，并不一定会写到硬盘中。并不能保证真正的持久化，但已经足够应付我们的简单工作队列。如果你一定要保证持久化，你需要改写你的代码来支持事务（transaction）。

## 公平调度

你应该已经发现，它仍旧没有按照我们期望的那样进行分发。比如有两个工作者（workers），处理奇数消息的比较繁忙，处理偶数消息的比较轻松。然而RabbitMQ并不知道这些，它仍然一如既往的派发消息。

这时因为RabbitMQ只管分发进入队列的消息，不会关心有多少消费者（consumer）没有作出响应。它盲目的把第n-th条消息发给第n-th个消费者。



我们可以使用basic.qos方法，并设置prefetch\_count=1。这样是告诉RabbitMQ，再同一时刻，不要发送超过1条消息给一个工作者（worker），直到它已经处理了上一条消息并且作出了响应。这样，RabbitMQ就会把消息分发给下一个空闲的工作者（worker）。

```
1. channel.basic_qos(prefetch_count=1)
```

## 关于队列大小

如果所有的工作者都处理繁忙状态，你的队列就会被填满。你需要留意这个问题，要么添加更多的工作者（*workers*），要么使用其他策略。

# 整合代码

`new_task.py`的完整代码：

```
1. #!/usr/bin/env python
2. import pika
3. import sys
4.
5. connection = pika.BlockingConnection(pika.ConnectionParameters(
6.     host='localhost'))
7. channel = connection.channel()
8.
9. channel.queue_declare(queue='task_queue', durable=True)
10.
11. message = ' '.join(sys.argv[1:]) or "Hello World!"
12. channel.basic_publish(exchange='',
13.                        routing_key='task_queue',
14.                        body=message,
15.                        properties=pika.BasicProperties(
16.                            delivery_mode = 2, # make message persistent
17.                        ))
18. print " [x] Sent %r" % (message,)
19. connection.close()
```

(`new_task.py`源码)

我们的worker：

```
1. #!/usr/bin/env python
2. import pika
3. import time
4.
5. connection = pika.BlockingConnection(pika.ConnectionParameters(
6.     host='localhost'))
7. channel = connection.channel()
8.
9. channel.queue_declare(queue='task_queue', durable=True)
10. print ' [*] Waiting for messages. To exit press CTRL+C'
11.
12. def callback(ch, method, properties, body):
13.     print " [x] Received %r" % (body,)
14.     time.sleep( body.count('.') )
15.     print " [x] Done"
16.     ch.basic_ack(delivery_tag = method.delivery_tag)
17.
18. channel.basic_qos(prefetch_count=1)
19. channel.basic_consume(callback,
```

```
20.         queue='task_queue')
21.
22. channel.start_consuming()
```

([worker.py](#) source)

使用消息响应和`prefetch_count`你就可以搭建起一个工作队列了。这些持久化的选项使得在RabbitMQ重启之后仍然能够恢复。

现在我们可以移步教程3学习如何发送相同的消息给多个消费者（consumers）。

# 发布/订阅

原文: [Publish/Subscribe](#)

状态: 校对完毕

翻译: [Adam](#)

校对: [Ping](#)

## 发布 / 订阅

(使用 **pika 0.9.5** Python客户端)

在上篇教程中，我们搭建了一个工作队列，每个任务只分发给一个工作者 (worker)。在本篇教程中，我们要做的跟之前完全不一样 —— 分发一个消息给多个消费者 (consumers)。这种模式被称为“发布 / 订阅”。

为了描述这种模式，我们将会构建一个简单的日志系统。它包括两个程序——第一个程序负责发送日志消息，第二个程序负责获取消息并输出内容。

在我们的这个日志系统中，所有正在运行的接收方程序都会接受消息。我们用其中一个接收者 (receiver) 把日志写入硬盘中，另外一个接受者 (receiver) 把日志输出到屏幕上。

最终，日志消息被广播给所有的接受者 (receivers)。

## 交换机 (Exchanges)

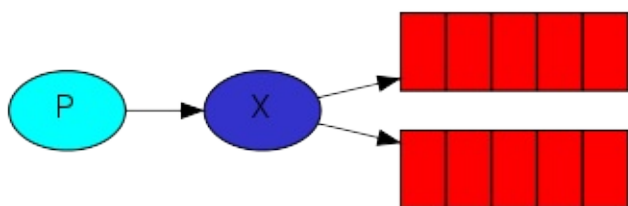
前面的教程中，我们发送消息到队列并从中取出消息。现在是时候介绍RabbitMQ中完整的消息模型了。

让我们简单的概括一下之前的教程：

- 发布者 (producer) 是发布消息的应用程序。
- 队列 (queue) 用于消息存储的缓冲。
- 消费者 (consumer) 是接收消息的应用程序。

RabbitMQ消息模型核心理念是：发布者 (producer) 不会直接发送任何消息给队列。事实上，发布者 (producer) 甚至不知道消息是否已经被投递到队列。

发布者 (producer) 只需要把消息发送给一个交换机 (exchange)。交换机非常简单，它一边从发布者方接收消息，一边把消息推送到队列。交换机必须知道如何处理它接收到的消息，是应该推送到指定的队列还是多个队列，或者是直接忽略消息。这些规则是通过交换机类型 (exchange type) 来定义的。



有几个可供选择的交换机类型：直连交换机 (direct)，主题交换机 (topic)，(头交换机) headers 和 扇型交换机 (fanout)。我们在这里主要说明最后一个 —— 扇型交换机 (fanout)。先创建一个fanout类型的交换机，命名为logs：

```
1. channel.exchange_declare(exchange='logs',
2.                           type='fanout')
```

扇型交换机（fanout）很简单，你可能从名字上就能猜测出来，它把消息发送给它所知道的所有队列。这正是我们的日志系统所需要的。

## 交换器列表

`rabbitmqctl`能够列出服务器上所有的交换器：

```
1. $ sudo rabbitmqctl list_exchanges
2. Listing exchanges ...
3. logs      fanout
4. amq.direct direct
5. amq.topic  topic
6. amq.fanout fanout
7. amq.headers headers
8. ...done.
```

这个列表中有一些叫做`amq.*`的交换器。这些都是默认创建的，不过这时候你还不需要使用他们。

## 匿名的交换器

前面的教程中我们对交换机一无所知，但仍然能够发送消息到队列中。因为我们使用了命名为空字符串（`""`）默认的交换机。

回想我们之前是如何发布一则消息：

```
1. channel.basic_publish(exchange='',
2.                       routing_key='hello',
3.                       body=message)
```

`exchange`参数就是交换机的名称。空字符串代表默认或者匿名交换机：消息将会根据指定的`routing_key`分发到指定的队列。

现在，我们就可以发送消息到一个具名交换机了：

```
1. channel.basic_publish(exchange='logs',
2.                       routing_key='',
3.                       body=message)
```

# 临时队列

你还记得之前我们使用的队列名吗（`hello`和`task_queue`）？给一个队列命名是很重要的——我们需要把工作（`workers`）指向正确的队列。如果你打算在发布者（`producers`）和消费者（`consumers`）之间共享同队列的话，给队列命名是十分重要的。

但是这并不适用于我们的日志系统。我们打算接收所有的日志消息，而不仅仅是一小部分。我们关心的是最新的消息而不是旧的。为了解决这个问题，我们需要做两件事情。

首先，当我们连接上RabbitMQ的时候，我们需要一个全新的、空的队列。我们可以手动创建一个随机的队列名，或者让服务器为我们选择一个随机的队列名（推荐）。我们只需要在调用`queue_declare`方法的时候，不提供`queue`参数

就可以了：

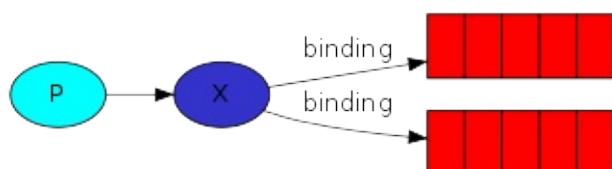
```
1. result = channel.queue_declare()
```

这时候我们可以通过`result.method.queue`获得已经生成的随机队列名。它可能是这样子的：`amq.gen-U0srCow8TsaXjNh73pnVAw==`。

第二步，当与消费者（consumer）断开连接的时候，这个队列应当被立即删除。`exclusive`标识符即可达到此目的。

```
1. result = channel.queue_declare(exclusive=True)
```

## 绑定（Bindings）



我们已经创建了一个扇型交换机（fanout）和一个队列。现在我们需要告诉交换机如何发送消息给我们的队列。交换器和队列之间的联系我们称之为绑定（binding）。

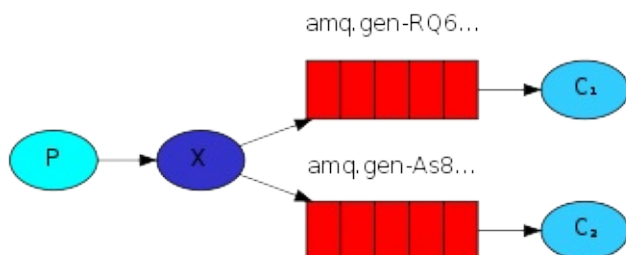
```
1. channel.queue_bind(exchange='logs',
2.                     queue=result.method.queue)
```

现在，logs交换机将会把消息添加到我们的队列中。

### 绑定（binding）列表

你可以使用 `rabbitmqctl list_bindings` 列出所有现存的绑定。

## 代码整合



发布日志消息的程序看起来和之前的没有太大区别。最重要的改变就是我们把消息发送给logs交换机而不是匿名交换机。在发送的时候我们需要提供`routing_key`参数，但是它的值会被扇型交换机（fanout exchange）忽略。以下是`emit_log.py`脚本：

```

1. #!/usr/bin/env python
2. import pika
3. import sys
4.
5. connection = pika.BlockingConnection(pika.ConnectionParameters(
6.     host='localhost'))
7. channel = connection.channel()
8.
9. channel.exchange_declare(exchange='logs',
10.     exchange_type='fanout')
11.
12. message = ' '.join(sys.argv[1:]) or "info: Hello World!"
13. channel.basic_publish(exchange='logs',
14.     routing_key='',
15.     body=message)
16. print " [x] Sent %r" % (message,)
17. connection.close()

```

(emit\_log.py 源文件)

正如你看到的那样，在连接成功之后，我们声明了一个交换器，这一个是重要的，因为不允许发布消息到不存在的交换器。

如果没有绑定队列到交换器，消息将会丢失。但这个没有所谓，如果没有消费者监听，那么消息就会被忽略。

receive\_logs.py的代码：

```

1. #!/usr/bin/env python
2. import pika
3.
4. connection = pika.BlockingConnection(pika.ConnectionParameters(
5.     host='localhost'))
6. channel = connection.channel()
7.
8. channel.exchange_declare(exchange='logs',
9.     exchange_type='fanout')
10.
11. result = channel.queue_declare(exclusive=True)
12. queue_name = result.method.queue
13.
14. channel.queue_bind(exchange='logs',
15.     queue=queue_name)
16.
17. print ' [*] Waiting for logs. To exit press CTRL+C'
18.
19. def callback(ch, method, properties, body):
20.     print " [x] %r" % (body,)
21.
22. channel.basic_consume(callback,
23.     queue=queue_name,
24.     no_ack=True)
25.

```



```
26. channel.start_consuming()
```

([receive\\_logs.py](#) source)

这样我们就完成了。如果你想把日志保存到文件中，只需要打开控制台输入：

```
1. $ python receive_logs.py > logs_from_rabbit.log
```

如果你想在屏幕中查看日志，那么打开一个新的终端然后运行：

```
1. $ python receive_logs.py
```

当然还要发送日志：

```
1. $ python emit_log.py
```

使用 `rabbitmqctl list_bindings` 你可确认已经创建的队列绑定。你可以看到运行中的两个`receive_logs.py`程序：

```
1. $ sudo rabbitmqctl list_bindings
2. Listing bindings ...
3. ...
4. logs      amq.gen-TJWkez28YpImbWdRKMa8sg==      []
5. logs      amq.gen-x0kymA4yPzAT6BoC/YP+zw==      []
6. ...done.
```

显示结果很直观：logs交换器把数据发送给两个系统命名的队列。这就是我们所期望的。

如何监听消息的子集呢？让我们移步教程4

# 路由

原文: [Routing](#)

状态: 翻译完成

翻译: [Adam](#)

校对: [Ping](#)

## 路由(Routing)

(使用 **pika 0.9.5** Python客户端)

在前面的教程中，我们实现了一个简单的日志系统。可以把日志消息广播给多个接收者。

本篇教程中我们打算新增一个功能 — 使得它能够只订阅消息的一个字集。例如，我们只需要把严重的错误日志信息写入日志文件（存储到磁盘），但同时仍然把所有的日志信息输出到控制台中

## 绑定 (Bindings)

前面的例子，我们已经创建过绑定 (bindings)，代码如下：

```
1. channel.queue_bind(exchange=exchange_name,  
2.                    queue=queue_name)
```

绑定 (binding) 是指交换机 (exchange) 和队列 (queue) 的关系。可以简单理解为：这个队列 (queue) 对这个交换机 (exchange) 的消息感兴趣。

绑定的时候可以带上一个额外的 `routing_key` 参数。为了避免与 `basic_publish` 的参数混淆，我们把它叫做绑定键 (binding key)。以下是如何创建一个带绑定键的绑定。

```
1. channel.queue_bind(exchange=exchange_name,  
2.                    queue=queue_name,  
3.                    routing_key='black')
```

绑定键的意义取决于交换机 (exchange) 的类型。我们之前使用过的扇型交换机 (fanout exchanges) 会忽略这个值。

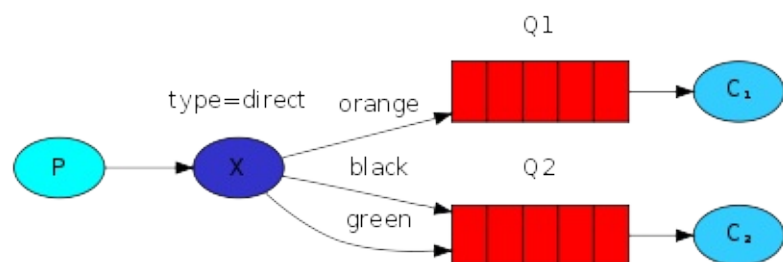
## 直连交换机 (Direct exchange)

我们的日志系统广播所有的消息给所有的消费者 (consumers)。我们打算扩展它，使其基于日志的严重程度进行消息过滤。例如我们也许只是希望将比较严重的错误 (error) 日志写入磁盘，以免在警告 (warning) 或者信息 (info) 日志上浪费磁盘空间。

我们使用的扇型交换机 (fanout exchange) 没有足够的灵活性 — 它能做的仅仅是广播。

我们将会使用直连交换机 (direct exchange) 来代替。路由的算法很简单 — 交换机将会对绑定键 (binding key) 和路由键 (routing key) 进行精确匹配，从而确定消息该分发到哪个队列。

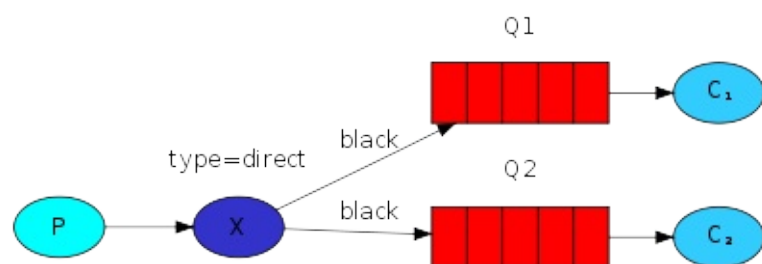
下图能够很好的描述这个场景：



在这个场景中，我们可以看到直连交换机 X和两个队列进行了绑定。第一个队列使用orange作为绑定键，第二个队列有两个绑定，一个使用black作为绑定键，另外一个使用green。

这样以来，当路由键为orange的消息发布到交换机，就会被路由到队列Q1。路由键为black或者green的消息就会路由到Q2。其他的所有消息都将会被丢弃。

## 多个绑定 (Multiple bindings)



多个队列使用相同的绑定键是合法的。这个例子中，我们可以添加一个X和Q1之间的绑定，使用black绑定键。这样一来，直连交换机就和扇型交换机的行为一样，会将消息广播到所有匹配的队列。带有black路由键的消息会同时发送到Q1和Q2。

## 发送日志

我们将会发送消息到一个直连交换机，把日志级别作为路由键。这样接收日志的脚本就可以根据严重级别来选择它想要处理的日志。我们先看看发送日志。

我们需要创建一个交换机 (exchange)：

```

1. channel.exchange_declare(exchange='direct_logs',
2.                           type='direct')
  
```

然后我们发送一则消息：

```

1. channel.basic_publish(exchange='direct_logs',
2.                       routing_key=severity,
3.                       body=message)
  
```

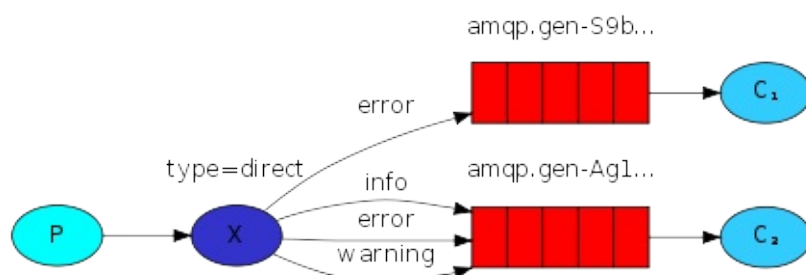
我们先假设“severity”的值是info、warning、error中的一个。

## 订阅

处理接收消息的方式和之前差不多，只有一个例外，我们将会为我们感兴趣的每个严重级别分别创建一个新的绑定。

```
1. result = channel.queue_declare(exclusive=True)
2. queue_name = result.method.queue
3.
4. for severity in severities:
5.     channel.queue_bind(exchange='direct_logs',
6.                         queue=queue_name,
7.                         routing_key=severity)
```

## 代码整合



emit\_log\_direct.py的代码：

```
1. #!/usr/bin/env python
2. import pika
3. import sys
4.
5. connection = pika.BlockingConnection(pika.ConnectionParameters(
6.     host='localhost'))
7. channel = connection.channel()
8.
9. channel.exchange_declare(exchange='direct_logs',
10.                           type='direct')
11.
12. severity = sys.argv[1] if len(sys.argv) > 1 else 'info'
13. message = ' '.join(sys.argv[2:]) or 'Hello World!'
14. channel.basic_publish(exchange='direct_logs',
15.                       routing_key=severity,
16.                       body=message)
17. print " [x] Sent %r:%r" % (severity, message)
18. connection.close()
```

receive\_logs\_direct.py的代码:

```

1. #!/usr/bin/env python
2. import pika
3. import sys
4.
5. connection = pika.BlockingConnection(pika.ConnectionParameters(
6.     host='localhost'))
7. channel = connection.channel()
8.
9. channel.exchange_declare(exchange='direct_logs',
10.     type='direct')
11.
12. result = channel.queue_declare(exclusive=True)
13. queue_name = result.method.queue
14.
15. severities = sys.argv[1:]
16. if not severities:
17.     print >> sys.stderr, "Usage: %s [info] [warning] [error]" % \
18.         (sys.argv[0],)
19.     sys.exit(1)
20.
21. for severity in severities:
22.     channel.queue_bind(exchange='direct_logs',
23.         queue=queue_name,
24.         routing_key=severity)
25.
26. print ' [*] Waiting for logs. To exit press CTRL+C'
27.
28. def callback(ch, method, properties, body):
29.     print " [x] %r:%r" % (method.routing_key, body,)
30.
31. channel.basic_consume(callback,
32.     queue=queue_name,
33.     no_ack=True)
34.
35. channel.start_consuming()

```

如果你希望只是保存warning和error级别的日志到磁盘，只需要打开控制台并输入：

```
1. $ python receive_logs_direct.py warning error > logs_from_rabbit.log
```

如果你希望所有的日志信息都输出到屏幕中，打开一个新的终端，然后输入：

```

1. $ python receive_logs_direct.py info warning error
2.  [*] Waiting for logs. To exit press CTRL+C

```

如果要触发一个error级别的日志，只需要输入：

```
1. $ python emit_log_direct.py error "Run. Run. Or it will explode."
```

```
2. [x] Sent 'error': 'Run. Run. Or it will explode.'
```

这里是完整的代码：([emit\\_log\\_direct.py](#)和[receive\\_logs\\_direct.py](#))

# 主题交换机

原文: [Topics](#)  
状态: 翻译完成  
翻译: [Ping](#)  
校对: [Ping](#)

## 为什么需要主题交换机？

(使用Python 客户端 — pika 0.9.8)

上一篇教程里，我们改进了我们的日志系统。我们使用直连交换机替代了扇型交换机，从只能盲目的广播消息改进为有可能选择性的接收日志。

尽管直连交换机能够改善我们的系统，但是它也有它的限制 — 没办法基于多个标准执行路由操作。

在我们的日志系统中，我们不只希望订阅基于严重程度的日志，同时还希望订阅基于发送来源的日志。Unix工具syslog就是同时基于严重程度-severity (info/warn/crit...) 和 设备-facility (auth/cron/kern...) 来路由日志的。

如果这样的话，将会给予我们非常大的灵活性，我们既可以监听来源于“cron”的严重程度为“critical errors”的日志，也可以监听来源于“kern”的所有日志。

为了实现这个目的，接下来我们学习如何使用另一种更复杂的交换机 — 主题交换机。

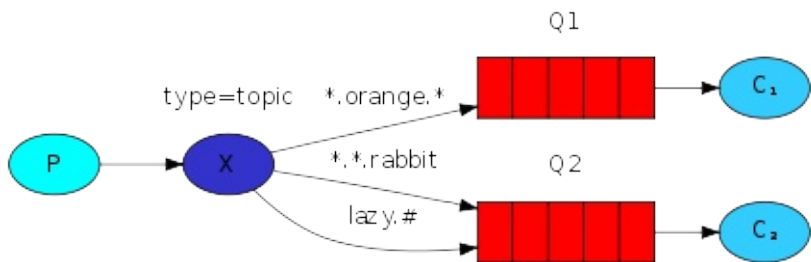
## 主题交换机

发送到主题交换机 (topic exchange) 的消息不可以携带随意什么样子的路由键 (routing\_key)，它的路由键必须是一个由 `.` 分隔开的词语列表。这些单词随便是什么都可以，但是最好是跟携带它们的消息有关系的词汇。以下是几个推荐的例子：“stock.usd.nyse”，“nyse.vmw”，“quick.orange.rabbit”。词语的个数可以随意，但是不要超过255字节。

绑定键也必须拥有同样的格式。主题交换机背后的逻辑跟直连交换机很相似 — 一个携带着特定路由键的消息会被主题交换机投递给绑定键与之想匹配的队列。但是它的绑定键和路由键有两个特殊应用方式：

- `*` (星号) 用来表示一个单词。
- `#` (井号) 用来表示任意数量 (零个或多个) 单词。

下边用图说明：



这个例子里，我们发送的所有消息都是用来描述小动物的。发送的消息所携带的路由键是由三个单词所组成的，这三个单词被两个 `.` 分割开。路由键里的第一个单词描述的是动物的手脚的利索程度，第二个单词是动物的颜色，第三个是动物的种类。所以它看起来是这样的：`<celerity>.<colour>.<species>`。

我们创建了三个绑定：Q1的绑定键为 `*.orange.*`，Q2的绑定键为 `*.*.rabbit` 和 `lazy.#`。

这三个绑定键被可以总结为：

- Q1 对所有的桔黄色动物都感兴趣。
- Q2 则是对所有的兔子和所有懒惰的动物感兴趣。

一个携带有 `quick.orange.rabbit` 的消息将会被分别投递给这两个队列。携带着 `lazy.orange.elephant` 的消息同样也会给两个队列都投递过去。另一方面携带有 `quick.orange.fox` 的消息会投递给第一个队列，携带有 `lazy.brown.fox` 的消息会投递给第二个队列。携带有 `lazy.pink.rabbit` 的消息只会被投递给第二个队列一次，即使它同时匹配第二个队列的两个绑定。携带着 `quick.brown.fox` 的消息不会投递给任何一个队列。

如果我们违反约定，发送了一个携带有一个单词或者四个单词（`"orange"` or `"quick.orange.male.rabbit"`）的消息时，发送的消息不会投递给任何一个队列，而且会丢失掉。

但是另一方面，即使 `"lazy.orange.male.rabbit"` 有四个单词，他还是会匹配最后一个绑定，并且被投递到第二个队列中。

## 主题交换机

主题交换机是很强大的，它可以表现出跟其他交换机类似的行为

当一个队列的绑定键为 `"#"`（井号）的时候，这个队列将会无视消息的路由键，接收所有的消息。

当 `*`（星号）和 `#`（井号）这两个特殊字符都未在绑定键中出现的时候，此时主题交换机就拥有的直连交换机的行为。

## 组合在一起

接下来我们会将主题交换机应用到我们的日志系统中。在开始工作前，我们假设日志的路由键由两个单词组成，路由键看起来是这样的：`<facility>.<severity>`

代码跟[上一篇教程](#)差不多。

emit\_log\_topic.py的代码：

```
1. #!/usr/bin/env python
2. import pika
3. import sys
4.
5. connection = pika.BlockingConnection(pika.ConnectionParameters(
6.     host='localhost'))
7. channel = connection.channel()
8.
9. channel.exchange_declare(exchange='topic_logs',
10.                           type='topic')
11.
12. routing_key = sys.argv[1] if len(sys.argv) > 1 else 'anonymous.info'
13. message = ' '.join(sys.argv[2:]) or 'Hello World!'
```



```

14. channel.basic_publish(exchange='topic_logs',
15.                        routing_key=routing_key,
16.                        body=message)
17. print " [x] Sent %r:%r" % (routing_key, message)
18. connection.close()

```

receive\_logs\_topic.py的代码:

```

1. #!/usr/bin/env python
2. import pika
3. import sys
4.
5. connection = pika.BlockingConnection(pika.ConnectionParameters(
6.     host='localhost'))
7. channel = connection.channel()
8.
9. channel.exchange_declare(exchange='topic_logs',
10.                          type='topic')
11.
12. result = channel.queue_declare(exclusive=True)
13. queue_name = result.method.queue
14.
15. binding_keys = sys.argv[1:]
16. if not binding_keys:
17.     print >> sys.stderr, "Usage: %s [binding_key]..." % (sys.argv[0],)
18.     sys.exit(1)
19.
20. for binding_key in binding_keys:
21.     channel.queue_bind(exchange='topic_logs',
22.                       queue=queue_name,
23.                       routing_key=binding_key)
24.
25. print ' [*] Waiting for logs. To exit press CTRL+C'
26.
27. def callback(ch, method, properties, body):
28.     print " [x] %r:%r" % (method.routing_key, body,)
29.
30. channel.basic_consume(callback,
31.                       queue=queue_name,
32.                       no_ack=True)
33.
34. channel.start_consuming()

```

执行下边命令 接收所有日志:

```
python receive_logs_topic.py "#"
```

执行下边命令 接收来自“kern”设备的日志:

```
python receive_logs_topic.py "kern.*"
```

执行下边命令 只接收严重程度为“critical”的日志:

```
python receive_logs_topic.py "/*.critical"
```

执行下边命令 建立多个绑定：

```
python receive_logs_topic.py "kern.*" "/*.critical"
```

执行下边命令 发送路由键为 “kern.critical” 的日志：

```
python emit_log_topic.py "kern.critical" "A critical kernel error"
```

执行上边命令试试看效果吧。另外，上边代码不会对路由键和绑定键做任何假设，所以你可以在命令中使用超过两个路由键参数。

## 如果你现在还没被搞晕，想想下边问题：

- 绑定键为 `*` 的队列会取到一个路由键为空的消息吗？
- 绑定键为 `#.*` 的队列会获取到一个名为 `..` 的路由键的消息吗？它会取到一个路由键为单个单词的消息吗？
- `a.*.#` 和 `a.##` 的区别在哪儿？

(完整代码参见[emit\\_logs\\_topic.py](#) and [receive\\_logs\\_topic.py](#))

移步至[教程 6](#) 学习RPC。

# 远程过程调用

原文: [Remote procedure call \(RPC\)](#)

状态: 翻译完成

翻译: [Ping](#)

校对: [Ping](#)

## 远程过程调用 (RPC)

### (Python客户端 — 使用 pika 0.9.8)

在[第二篇教程](#)中我们介绍了如何使用工作队列 (work queue) 在多个工作者 (worker) 中间分发耗时的任务。

可是如果我们需要将一个函数运行在远程计算机上并且等待从那儿获取结果时, 该怎么办呢? 这就是另外的故事了。这种模式通常被称为远程过程调用 (Remote Procedure Call) 或者RPC。

这篇教程中, 我们会使用RabbitMQ来构建一个RPC系统: 包含一个客户端和一个RPC服务器。现在的情况是, 我们没有一个值得被分发的足够耗时的任务, 所以接下来, 我们会创建一个模拟RPC服务来返回斐波那契数列。

## 客户端接口

为了展示RPC服务如何使用, 我们创建了一个简单的客户端类。它会暴露出一个名为“call”的方法用来发送一个RPC请求, 并且在收到回应前保持阻塞。

```
1. fibonacci_rpc = FibonacciRpcClient()
2. result = fibonacci_rpc.call(4)
3. print "fib(4) is %r" % (result,)
```

### 关于RPC的注意事项:

尽管RPC在计算领域是一个常用模式, 但它也经常被告病。当一个问题被抛出的时候, 程序员往往意识不到这到底是由本地调用还是由较慢的RPC调用引起的。同样的困惑还来自于系统的不可预测性和给调试工作带来的不必要的复杂性。跟软件精简不同的是, 滥用RPC会导致不可维护的[面条代码](#)。

考虑到这一点, 牢记以下建议:

确保能够明确的搞清楚哪个函数是本地调用的, 哪个函数是远程调用的。给你的系统编写文档。保持各个组件间的依赖明确。处理错误案例。明了客户端如何处理RPC服务器的宕机和长时间无响应情况。

当对避免使用RPC有疑问的时候。如果可以的话, 你应该尽量使用异步管道来代替RPC类的阻塞。结果被异步地推送到下一个计算场景。

## 回调队列

一般来说通过RabbitMQ来实现RPC是很容易的。一个客户端发送请求信息, 服务器端将其应用到一个回复信息中。为了接收到回复信息, 客户端需要在发送请求的时候同时发送一个回调队列 (callback queue) 的地址。我们试试看:

```
1. result = channel.queue_declare(exclusive=True)
2. callback_queue = result.method.queue
3.
4. channel.basic_publish(exchange='',
```

```

5.         routing_key='rpc_queue',
6.         properties=pika.BasicProperties(
7.             reply_to = callback_queue,
8.         ),
9.         body=request)
10.
11. # ... and some code to read a response message from the callback_queue ...

```

## 消息属性

AMQP协议给消息预定义了一系列的14个属性。大多数属性很少会用到，除了以下几个：

- *delivery\_mode* (投递模式)：将消息标记为持久的（值为2）或暂存的（除了2之外的其他任何值）。第二篇教程里接触过这个属性，记得吧？
- *content\_type* (内容类型)：用来描述编码的*mime-type*。例如在实际使用中常常使用*application/json*来描述JSON编码类型。
- *reply\_to* (回复目标)：通常用来命名回调队列。
- *correlation\_id* (关联标识)：用来将RPC的响应和请求关联起来。

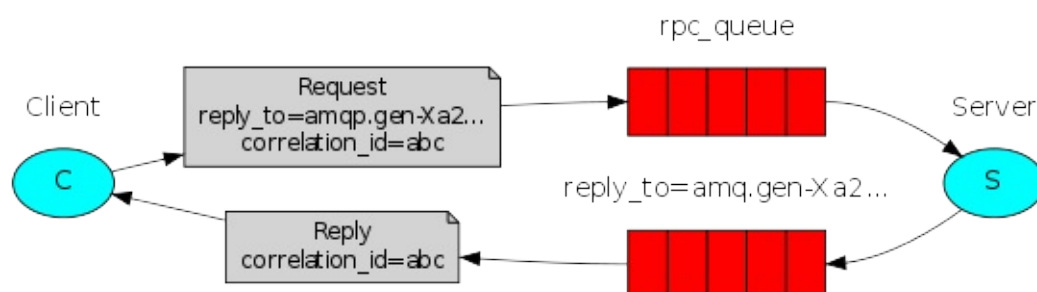
## 关联标识

上边介绍的方法中，我们建议给每一个RPC请求新建一个回调队列。这不是一个高效的办法，幸好这儿有一个更好的办法——我们可以为每个客户端只建立一个独立的回调队列。

这就带来一个新问题，当此队列接收到一个响应的时候它无法辨别出这个响应是属于哪个请求的。**correlation\_id**就是为了解决这个问题而来的。我们给每个请求设置一个独一无二的值。稍后，当我们从回调队列中接收到一个消息的时候，我们就可以查看这条属性从而将响应和请求匹配起来。如果我们接手到的消息的*correlation\_id*是未知的，那就直接销毁掉它，因为它不属于我们的任何一条请求。

你也许会问，为什么我们接收到未知消息的时候不抛出一个错误，而是要将它忽略掉？这是为了解决服务器端有可能发生的竞争情况。尽管可能性不大，但RPC服务器还是有可能在已将应答发送给我们但还未将确认消息发送给请求的情况下死掉。如果这种情况发生，RPC在重启后会重新处理请求。这就是为什么我们必须在客户端优雅的处理重复响应，同时RPC也需要尽可能保持幂等性。

## 总结



我们的RPC如此工作：

- 当客户端启动的时候，它创建一个匿名独享的回调队列。
- 在RPC请求中，客户端发送带有两个属性的消息：一个是设置回调队列的 *reply\_to* 属性，另一个是设置唯一值的 *correlation\_id* 属性。

- 将请求发送到一个 `rpc_queue` 队列中。
- RPC工作者（又名：服务器）等待请求发送到这个队列中来。当请求出现的时候，它执行他的工作并且将带有执行结果的消息发送给`reply_to`字段指定的队列。
- 客户端等待回调队列里的数据。当有消息出现的时候，它会检查`correlation_id`属性。如果此属性的值与请求匹配，将它返回给应用。

## 整合到一起

`rpc_server.py`代码：

```

1. #!/usr/bin/env python
2. import pika
3.
4. connection = pika.BlockingConnection(pika.ConnectionParameters(
5.     host='localhost'))
6.
7. channel = connection.channel()
8.
9. channel.queue_declare(queue='rpc_queue')
10.
11. def fib(n):
12.     if n == 0:
13.         return 0
14.     elif n == 1:
15.         return 1
16.     else:
17.         return fib(n-1) + fib(n-2)
18.
19. def on_request(ch, method, props, body):
20.     n = int(body)
21.
22.     print " [.] fib(%s)" % (n,)
23.     response = fib(n)
24.
25.     ch.basic_publish(exchange='',
26.                     routing_key=props.reply_to,
27.                     properties=pika.BasicProperties(correlation_id = \
28.                                                     props.correlation_id),
29.                     body=str(response))
30.     ch.basic_ack(delivery_tag = method.delivery_tag)
31.
32. channel.basic_qos(prefetch_count=1)
33. channel.basic_consume(on_request, queue='rpc_queue')
34.
35. print " [x] Awaiting RPC requests"
36. channel.start_consuming()
```

服务器端代码相当简单：

- （4）像往常一样，我们建立连接，声明队列

- (11) 我们声明我们的fibonacci函数，它假设只有合法的正整数当作输入。（别指望这个函数能处理很大的数值，函数递归你们都懂得...）
- (19) 我们为 basic\_consume 声明了一个回调函数，这是RPC服务器端的核心。它执行实际的操作并且作出响应。
- (32) 或许我们希望能在服务器上多开几个线程。为了能将负载平均地分摊到多个服务器，我们需要将 prefetch\_count 设置好。

rpc\_client.py 代码：

```

1. #!/usr/bin/env python
2. import pika
3. import uuid
4.
5. class FibonacciRpcClient(object):
6.     def __init__(self):
7.         self.connection = pika.BlockingConnection(pika.ConnectionParameters(
8.             host='localhost'))
9.
10.        self.channel = self.connection.channel()
11.
12.        result = self.channel.queue_declare(exclusive=True)
13.        self.callback_queue = result.method.queue
14.
15.        self.channel.basic_consume(self.on_response, no_ack=True,
16.                                    queue=self.callback_queue)
17.
18.    def on_response(self, ch, method, props, body):
19.        if self.corr_id == props.correlation_id:
20.            self.response = body
21.
22.    def call(self, n):
23.        self.response = None
24.        self.corr_id = str(uuid.uuid4())
25.        self.channel.basic_publish(exchange='',
26.                                   routing_key='rpc_queue',
27.                                   properties=pika.BasicProperties(
28.                                       reply_to = self.callback_queue,
29.                                       correlation_id = self.corr_id,
30.                                   ),
31.                                   body=str(n))
32.        while self.response is None:
33.            self.connection.process_data_events()
34.        return int(self.response)
35.
36. fibonacci_rpc = FibonacciRpcClient()
37.
38. print " [x] Requesting fib(30)"
39. response = fibonacci_rpc.call(30)
40. print " [.] Got %r" % (response,)
```

客户端代码稍微有点难懂：

- (7) 建立连接、通道并且为回复 (replies) 声明独享的回调队列。
- (16) 我们订阅这个回调队列，以便接收RPC的响应。
- (18) “on\_response”回调函数对每一个响应执行一个非常简单的操作，检查每一个响应消息的 correlation\_id 属性是否与我们期待的一致，如果一致，将响应结果赋给 self.response，然后跳出 consuming 循环。
- (23) 接下来，我们定义我们的主要方法 call 方法。它执行真正的RPC请求。
- (24) 在这个方法中，首先我们生成一个唯一的 correlation\_id 值并且保存起来，‘on\_response’回调函数会用它来获取符合要求的响应。
- (25) 接下来，我们将带有 reply\_to 和 correlation\_id 属性的消息发布出去。
- (32) 现在我们可以坐下来，等待正确的响应到来。
- (33) 最后，我们将响应返回给用户。

我们的RPC服务已经准备就绪了，现在启动服务器端：

```
1. $ python rpc_server.py
2. [x] Awaiting RPC requests
```

运行客户端，请求一个fibonacci队列。

```
1. $ python rpc_client.py
2. [x] Requesting fib(30)
```

此处呈现的设计并不是实现RPC服务的唯一方式，但是他有一些重要的优势：

- 如果RPC服务器运行的过慢的时候，你可以通过运行另外一个服务器端轻松扩展它。试试在控制台中运行第二个 rpc\_server.py 。
- 在客户端，RPC请求只发送或接收一条消息。不需要像 queue\_declare 这样的异步调用。所以RPC客户端的单个请求只需要一个网络往返。

我们的代码依旧非常简单，而且没有试图去解决一些复杂（但是重要）的问题，如：

- 当没有服务器运行时，客户端如何作出反映。
- 客户端是否需要实现类似RPC超时的东西。
- 如果服务器发生故障，并且抛出异常，应该被转发到客户端吗？
- 在处理前，防止混入无效的信息（例如检查边界）

如果你想做一些实验，你会发现 [rabbitmq-management plugin](#) 在观测队列方面是很有用处的。

（完整的 [rpc\\_client.py](#) 和 [rpc\\_server.py](#) 代码）