Broadview
www.broadview.com.cn

MANNING

RabbitMQ实战
高效部署分布式消息队列

[美] Alvaro Videla
Jason J.W. Williams 著
汪佳南 译

RabbitMQ in Action
Distributed Messaging for Everyone

xis Richardson
荐力荐

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

RabbitMQ实战
高效部署分布式消息队列

[美] Alvaro Videla
Jason J.W. Williams 著
汪佳南 译

RabbitMQ in Action
Distributed Messaging for Everyone

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书对RabbitMQ做了全面、翔实的讲解，体现了两位专家的真知灼见。本书首先介绍了有关MQ的历史，然后从基本的消息通信原理讲起，带领读者一路探索RabbitMQ的消息通信世界。这当中不仅包含了针对单台RabbitMQ服务器和RabbitMQ集群的讲解，还教导读者如何使用各种工具来进行监控。

本书内容浅显易懂，文笔风趣幽默。书中包含了丰富的、可以运行的示例程序源代码，读者可以自行下载并运行，这种亲身实践有助于加深对相关主题的理解。对于初学者来说，这样的编排非常适用。对于那些有一定RabbitMQ使用经验的读者来说，本书不仅可以巩固消息通信基本原理，同时也可加深对RabbitMQ高级特性的理解，譬如如何编写RabbitMQ插件。相信处于RabbitMQ探索之旅不同阶段的读者都能从本书获益。

本书适合有一定工作经验的开发者或者架构师阅读，也可以作为RabbitMQ运维工作的参考手册。

版权贸易合同登记号　图字：01-2014-4907

图书在版编目（CIP）数据

RabbitMQ实战：高效部署分布式消息队列 /（美）维德拉（Videla,A.），（美）威廉姆斯（Williams,J.J.W.）著；汪佳南译. —北京：电子工业出版社，2015.10
书名原文：RabbitMQ in Action: Distributed Messaging for Everyone
ISBN 978-7-121-27164-9

Ⅰ. ①R… Ⅱ. ①维… ②威… ③汪… Ⅲ. ①保密通信 Ⅳ. ①TN918.6

中国版本图书馆CIP数据核字（2015）第224246号

策划编辑：张春雨
责任编辑：李云静
印　　刷：北京天宇星印刷厂
装　　订：三河市皇庄路通装订厂
出版发行：电子工业出版社
　　北京市海淀区万寿路173信箱　　邮编：100036
开　　本：787×980　1/16　　印张：20.75　　字数：395千字
版　　次：2015年10月第1版
印　　次：2015年10月第1次印刷
定　　价：75.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。
质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。
服务热线：（010）88258888。

译者序

笔者从事互联网行业已五年有余，经历过大大小小十几个项目，其中包括金融领域方面的交易系统、医疗护理信息系统，再到后来的淘宝第三方软件供应商。这些项目不仅为我带来了丰富的阅历，更让我认识到软件开发的根本就在于降低软件复杂性。因此，我们需要采用可复用的软件设计模式，选取合适的软件构件来搭建自己的系统。这当中就用到了消息队列。

各位读者可能对消息队列并不陌生。消息队列提供一个异步通信协议，消息的发送者不用一直等待，直到消息被成功处理；而是立即返回。消息被暂存于队列当中，对消息感兴趣的消费者会订阅消息并处理它们。那么，在什么样的场景下需要使用消息队列呢？其中之一就是系统集成。这些系统可能是公司的遗留系统，抑或是第三方维护的系统。集成的方法有很多，其中之一是让这些系统往数据库的某几张表写入数据，然后让新的系统定时地去轮询并取出数据进行处理。这已经具备了消息队列通信的雏形：其中的数据库表就是消息队列，它定义了消息的标准结构，并暂存待处理的消息。对于小型系统和简单的需求来讲，这样的解决方案没什么问题，并且足够应付了。但是随着日后业务需求的扩张，接入的系统越来越多，对于消息通信的需求也变得越来越复杂：

1. 你需要将同一条消息发送给多个感兴趣的系统，同时又需要让系统只接受符合特定模式的消息。

2. 你开始觉得系统当中的同步处理方式严重影响了吞吐量，你想把用户订单处

理、消息通知等都修改成异步处理的方式。

3. 你开始关注系统的高可用性。同时你还想实时观测系统当中订单消息的处理速率，以便及时应对峰值压力。

通过使用标准的消息队列系统就可以轻松处理以上问题。当然它们提供的功能远不止这些。市面上有很多开放源码的消息队列系统，包括 JBoss Messaging、JORAM、Apache ActiveMQ、Sun Open Message Queue、Apache Qpid、RabbitMQ，等等。其中就有本书讨论的 RabbitMQ。它实现了 AMQP 协议，并且遵循 Mozilla Public License 开源协议。它支持多种编程语言，可以方便地和 Spring 集成。最重要的一点是，使用消息队列并不是“杀鸡焉用牛刀”的问题，而是一种未雨绸缪，随着系统不断演进，你终将从消息队列中获益。

翻译一本书绝非易事。在此，我要感谢给予我帮助的赵震一和陈祥奎夫妇，你们是我的挚友与伯乐。感谢电子工业出版社的编辑张春雨和李云静，是你们的细心指导保证了本书的翻译质量。最后感谢我的女友，你是我的精神支柱，在你的陪伴和支持下，我得以全心全意地完成工作。

由于时间仓促，文中难免有不足之处，希望读者海涵，也希望各位能和我一样从中收获知识和快乐，谢谢大家。

汪佳南

2015 年 9 月于杭州

致我的祖父Maximiliano Godoy，

是你为我指引了人生的道路。谢谢。

——A.V.

致我的妈妈、爸爸和妹妹J'aime，

你们的爱、支持和信任

使我能得以攀越高峰……

致上帝，因为你的庇佑总能让我成功到达彼岸。

——J.W.

序

欢迎阅读本书。如果你像我一样，你很有可能在想：“我是否应该从头开始阅读呢？”唉，市面上的技术书籍琳琅满目，但并不是所有的图书都值得花费时间关注。所以请让我邀你继续往下阅读，看看下面这份说明是否合你的胃口：

- 你想要通过一种实用的方法来学习推技术、流数据，以及其他消息通信模式。
- 你想要获得专家级别的 RabbitMQ 技能，包括在设计方面和生产环境运行方面的最佳实践。

换句话说，本书不仅仅是一本 RabbitMQ 指南。它会为你讲解能覆盖多种使用场景的基本设计模式。它展现了为何越来越多的应用开始使用这些模式，以及哪些“可以做”、哪些“不能做”。

这些模式是怎样的呢？如果你曾经有想法将系统绘成一幅信息流或者网络图，而非栈的形式，那么你可能正在使用，或者正准备使用消息通信。你可能一直在考虑数据投递、非阻塞操作或者推送通知。或者，你想使用发布 / 订阅模式、异步处理或是工作队列。所有这些模式构成了众所周知的消息通信（messaging）。

消息通信是一种至关重要的能力：它使得软件应用可以相互连接并得以扩展。应用程序们既可以像一个巨大应用中的组件一样相互连接起来，也可以连接用户的设备和数据。消息通信从本质上来说是异步的，因为它通过分隔了数据的发送和接收来解耦应用。最奇妙的地方在于该连接模式可以在任何规模下以相同的方式工作。

规模是关键点。互联网无处不在，它作为应用程序投递的基础，使得规模在应

用程序设计中成为关键因素。我们已经无法停留在小规模的思考方式上。最近，术语“大数据”（big data）已经变得非常流行。不过相较于最近几年来说，所有事物都往大的方向发展了。

举例来说，移动连接设备的数量很快就将超过地球上人口的数量。当我编写本书时，Facebook 正准备 IPO。CTO Bret Taylor 说道：“如果 Mark Zuckerberg 在宿舍编写 Facebook 的时候这些技术就已经成熟的话，那么现在的 Facebook 就已经是一个移动应用了。”

花点时间想想。大多数应用曾经都是这样运作的：加载文档或者从数据库获取数据，做一些处理，并将结果写入磁盘。将来的应用程序会更像 Facebook：总是在线，并且托管在云端，在任何地方都能访问。输入和处理是连续且自动的，并即时输出用户想要的过滤信息流。

如果不采用一系列专门的设计模式的话，那么这些级别的自动化、可访问性和伸缩性都无从谈起。在本书中你就能学到这些模式。作为现代消息通信技术发起者之一，Derek Collison 将消息通信描述为“移动的数据”，这一点令人印象深刻。很难想象有什么应用是不需要移动数据的。因此，消息通信无处不在。

这本书可以让你立即上手。消息通信模式将以可运行代码的形式展现在你面前。作者也会帮助你在自己的系统上运行起来。在 Jason J. W. Williams 和 Alvaro Videla 的帮助下，你就能零距离接触那些拥有多年运行大规模 RabbitMQ 系统经验的专家了。本书是他们将经验分享给社区这一杰出工作的成果。

在对 RabbitMQ 有了一点感觉之后，不管你使用何种语言编写程序，都能很容易地在广大 RabbitMQ 用户社区里寻求到帮助并找到更多示例。这使得 RabbitMQ 成为能满足你消息通信需求的最佳选择。

希望你已经被我吊起了胃口，准备翻开下一页继续阅读。书中会有关于消息和 RabbitMQ 的一切，作者将会为你揭开它们的神秘“面纱”。

Alexis Richardson

Rabbit Technologies 公司创始人和前任 CEO

VMware 云应用平台高级总监

前言

编写本书就像是在探索 RabbitMQ 一样——遇到需要解决的问题，但是却不知道解决方案是什么。在 2010 年 5 月前，我们甚至都不认识对方。在过去的两年里，我俩都活跃在 RabbitMQ 社区，但是我们并未真正相遇过。之后有一天在与 Alexis Richardson（时任 Rabbit 的 CEO）的一次谈话中，我与 Alvaro 互相认识了对方，这使得你现在手中所捧的这本书成为可能。我们之间的共同之处在于，我们都期望将自己通过艰难困苦才掌握的所有 RabbitMQ 知识写下来。回到 2010 年，这些知识（并且现今大部分时候仍然如此）散落在网络上那些一知半解的博客文章和简洁的技术指引中。换句话说，我俩都想要编写本书。真心希望本书在我们着手使用 RabbitMQ 的两年前就已经存在了。

我俩并非传统的消息通信背景出身，这使得我们很快成为朋友，并在很大程度上形成了本书的基调；我们假定本书的读者是那些之前从未听说过队列或者绑定（binding）的人。事实上，当我俩探索 RabbitMQ 时，我们甚至不知道消息通信是什么，或者说我们不知道这就是自己所寻找问题的解决方案。我（Jason）的处境是公司需要一种方法来获取从顾客那里收到的垃圾邮件报告，并在不同于消息主流的带外数据（out-of-band）中对它们进行处理。而 Alvaro 的情况是，他们公司的社交网络成员通信系统在 200GB 数据库负载下无法正常工作。就像很多其他诉诸消息通信的那些人一样，我俩首先尝试的是使用数据库表来解决以队列为中心的问题。像确保只有一个应用实例消费任意特定队列条目这样的问题，让我们在数据库驱动的方案上的尝试备受折磨，逼迫我们寻找更好的方法。毕竟，我们认识到自己不会是软

件历史上第一个碰到这类问题的人。

我们这两个问题的解决方案竟然惊人地相似：一位在 Plaxo 的朋友告诉我看看“RabbitMQ 之类的东西”是否能解决我的以队列为中心的问题，同时在中国的一位 Alvaro 的 Erlang 同事则给了他相同的建议。相距半个地球，我俩以相同的方式发现了 RabbitMQ，为的是尝试解决几乎同样的问题！事实上，你正阅读有关 RabbitMQ 的书，很有可能是因为相似的挑战以相同的方式指引着你来发现 RabbitMQ。这也解释了为何 RabbitMQ 如此流行：当尝试扩大我们构建的软件规模时，它能轻而易举地解决那些挥之不去的分布式数据的基本问题。

我们希望本书能够帮助你通过使用 RabbitMQ 来更快、更轻松地设计这些挑战的解决方案，从而使你可以将更多的时间用于编写改变世界的软件，同时花费更少的时间来了解如何加速消息通信代理。在这一路上，也许 RabbitMQ 会给你带来一位了不起的合著者，你从没想过他会成为你终生的朋友[1]。本书是我们热爱编写软件的产物，同时也希望它能以你从未想象的方式来帮助你完成相同的事情。

Alvaro Videla
迪本多夫，瑞士
Jason J. W. Williams
爱达荷州博伊西市，美国

1 大家都说合著者的关系相比婚姻来说拥有更高的“离婚率”。这倒是一个不错的比较，因为合著一本书需要合著者不断地相互迁就和尊重，这样彼此才能生活在同一“屋檐”下。感谢上帝能让我撰写本书，同时能交到这么一位志同道合的朋友。

致　谢

虽然本书封面上只出现了两个名字，但是如果没有其他人的帮助，也就不会有这本书的存在了。首先，在我们开始编写本书的时候，我们想要感谢 RabbitMQ 的 CEO Alexis Richardson。如果没有他的引荐，Manning 就不会给我们这个机会，我们也就不会一起创作这本书了。同时，我们也要感谢他为本书作序。同样地，我们也要向 RabbitMQ 团队表达最真挚的感谢，感谢他们一直以来的帮助，以及对我们关于 Rabbit 细节之处提问的耐心解答。特别是，我们要感谢 Matthew Sackman 和 Matthias Radestock，如果没有他们两位的话，就不会有关于集群和 RabbitMQ 内部机制这几个章节了。

尤其是 RabbitMQ 团队的 Jerry Kuch，我们对他的感激之情无以言表。Jerry 自愿花费了无数时间来为每一章草稿进行准确的审校，包括对已完成的图书进行“官方”技术审校。每当由于我们经验不足而需要说明和建议时，他总能在 IM 上快速回复。作为 RabbitMQ 团队的接口人，他非常耐心且从不抱怨。如果你发现自己正在探索未知的 Rabbit 操作，可能就需要感谢 Jerry Kuch。他让本书更为出色；同时他也是一名优秀的工程师。

我们再怎么感谢 Manning 初级项目编辑 Maria Townsley 也不为过。Maria 帮助我们按计划写作。她忍受着我们的工作安排，以及时好时坏的原稿。最重要的是，她是我们的拥护者，并且任何对我们来说重要的资源她都努力争取。如果你满意本书的写作风格，那就感谢 Maria 吧，是她指引着我们做到这一点的。同时，我们也要感谢 Cynthia Kane，是她让我们完成最后几章并付诸印刷。当我们的工作方式太老套的时候，Cynthia 以最终项目编辑的身份加入进来。她适应了我们的工作方式，

并且将本书的工作当作自己从第一天起就加入进来那样看待。Cynthia 就像我们的三垒教练并护送我们回家。

最后，我们要感谢忠实的读者，他们从 Manning 的早期预览项目（MEAP）中购买了本书，同时也要感谢我们的审校们：Barry Alexander、P. David Pull、Bruce Snyder、Tony Garnock-Jones、James Williams、Patrick Lemiuex、Bruce Lowekamp、Carlton Gibson、Paul Grebenc、Richard Siddaway、Gordon Dickens、Gene Campbell、Karsten Strøbæk、Jeff Addison、David Dossot、Daniel Bretoi 和 Ben Rockwood。你们没有报酬。但是你们把本书当作自己的孩子一样，给我们提出了详细的反馈和深思熟虑的见解。正是由于你们的功劳，才让本书变得无比美好。谢谢你们！

Alvaro

我想要感谢我的太太 Silvana，在本书的写作过程中她一直支持着我。我已经记不得由于编写本书的原因，我们错过了多少部电影、多久没有到外面逛逛了。我能说的只是“多谢理解”。另外，我还要感谢我的母亲一直以来对我的信任。毕竟，编写本书是全家人的功劳。我还要感谢我在中国 The Netcircle 的朋友们，在那里我开始对 Rabbit 深深地着迷了，并且还一天到晚不停地打扰他们。最后，我要感谢 Jason。Manning 为我引荐了这位合著者，我们最后也成了好朋友。

Jason

我要感谢我的父母和我的妹妹在整个写作过程中给予我的支持和爱。他们信任我，并且敦促我前进。当我不想写作的时候，他们会把我推出门外到咖啡馆继续写作。他们一直相信我能够完成本书的编写（即便这看来遥不可及）。

在和父母一起创建的初创公司里，我很荣幸能称呼父母为合伙人。作为我的合伙人，我亏欠了他们以及 DigiTar 太多太多了。当写作需要插入到工作时间的时候，他们从未抱怨过。并且给予我很大的自由度来平衡写作和工作。如果没有我们的公司，我也不会有动力去探索 Rabbit，也不会因写一些博客教程而受邀编写本书了。DigiTar 带给我许多祝福和机会，这本书就是其中之一。

最后我要感谢 Alvaro。我从不知道还存在你这么一位朋友，情同手足，就像来自另一位母亲的亲兄弟一样。谢谢你，你是我收到的最意外的祝福。

关于本书

RabbitMQ 是一个开源的消息代理和队列服务器，用来通过普通协议在完全不同的应用之间共享数据，或者简单地将作业排队以便让分布式服务器进行处理。这无关乎项目的大小：RabbitMQ 能够适应你的需要。你想要使用 X 编程语言来为其中一个应用组件快速建立原型，并能确保在第二天就可将其切换到一门更高性能的编程语言上吗？通过解耦通信协议，RabbitMQ 可以帮你做到这一点。你是否需要即时为你的社交网站处理图片上传，同时轻而易举地增加或者删除服务器？你可以使用 Rabbit 队列来存放作业，并让服务器负责为你做负载均衡和作业分派。通过使用 RabbitMQ 可以又快又简单地解决这类问题。本书的目的就在于向你展示如何围绕消息通信来实现最佳架构。

编写应用程序相对简单，保障应用正常运行才是挑战的开始。别担心，本书会讲到 RabbitMQ 管理、集群、安全和监控的最佳实践，因此你同样能够学习到运维方面的知识。

最后，我们将深入介绍 RabbitMQ 的核心和内部细节，让你理解服务器使用的系统资源，以便在设计系统架构的时候可以做好容量规划。同时，你将学习如何通过安装插件和创建自己的插件来扩展服务器。将你的编辑器准备好，因为我们就要开始用 Python、PHP、Erlang、Java 和 C# 来编程了。

路线图

第 1 章解释 AMQP 协议的起源、RabbitMQ 的诞生，以及它能解决什么样的行

业问题。下一步，你将安装服务器并创建第一个 Hello World 程序，通过 RabbitMQ 来发送数据。

第 2 章，你将沉浸在消息通信的世界中。我们从基础概念讲起，再到如何将这些概念映射到 AMQP（该协议用于 RabbitMQ）。之后，你将学习消息持久化，以及消息的生命周期：从消息的发布开始，直到在网络的另一头进行消费。

第 3 章展示基本服务器管理。你将看到如何启动和停止节点、如何配置权限，以及如何获取服务器数据统计。我们会给你一些有用的提示来为服务器做故障排除。

第 4 章介绍消息通信模式和最佳实践。你将学习发后即忘(fire-and-forget)模型、RPC 架构等。

第 5 章开启了 RabbitMQ 集群和高可用性设置系列，一共三章。在此，你将在本机和物理服务器上设置 RabbitMQ 集群。你将学习如何升级 RabbitMQ 集群节点，以及如何使用镜像队列。

第 6 章介绍如何使用 HAProxy 为 RabbitMQ 服务器做负载均衡。你也会学到如何创建智能的消息通信客户端，知道如何在服务器发生故障的时候进行重连。

第 7 章介绍主 / 备服务器是如何工作的，并以此结束了高可用性系列的讨论。你也将学习 Shovel 插件，它允许 RabbitMQ 跨越数据中心复制数据。

第 8 章展示如何对 RabbitMQ 进行管理。你将学习 RabbitMQ Management 插件及其 Web 界面。这一章的内容不止于此，我们还将展示该插件所提供的 REST API 概要。

第 9 章与第 8 章相衔接，详细介绍 REST API。你将学到如何通过使用该 API 来完成绝大多数的管理任务。为应用程序配置用户和虚拟主机提供前所未有的便利。

第 10 章教你如何监控 RabbitMQ，包括使用 Nagios 检测、使用 AMQP 和 REST API 来监控服务器的内部状态。你将学会如何通过检测问题来防患于未然。

第 11 章详细介绍交换器的内部工作机制（RabbitMQ 使用的路由算法）。我们将深入了解消息通信结构使用的资源细节，看看可以从架构决策中预见到什么。我们也会通过介绍为应用程序启用 SSL 连接来讲解 RabbitMQ 的安全性。

第 12 章将以如何通过添加新的插件来扩展 RabbitMQ 的行为来结束整本书。这

些插件既可以是别人编写发布的，也可以是你自己创建的。

编码规范及下载

所有源代码都以等宽字体排版（像这样 `fixed-width font like this`），以示区别。许多清单都伴有代码注释以突出重要的概念。在某些情况下，清单之后会有编号项目来连接到说明。

由于 RabbitMQ 的最大优势之一在于将不同语言编写的应用程序胶合在一起，因此我们同时使用 Python 和 PHP 作为示例的主要编程语言（在附录中会额外夹带一些 .NET 和 Java 的代码）。不过，我们想让这些示例对于使用不同编程语言的读者来说尽可能广泛地适用。由于我们不可能将每个示例都用不同的编程语言重写一遍，因此我们将这些示例代码提交到了 Github 代码库，以便你也能贡献代码：https://github.com/rabbitinaction/sourcecode。

在官方 Github 代码库中，你将能找到本书中最新版本的示例代码。这些代码中的一部分已经由其他读者转换成了不同的编程语言，譬如 Ruby。找不到你喜欢的编程语言？ Fork 代码库，然后添加进去吧！之后发送给我们一个 pull request，我们会尽全力合并你的示例代码版本。（注意：你必须使用和我们的代码相同的 BSD 许可，以便我们将你的变更合并进来。）

如果你喜欢本书中权威且真正“官方”的示例代码的话，可以从出版商的网站中进行下载：http://www.manning.com/videla/。本书的最新版本中出现的具体代码总可以在这里找到。

关于作者

Alvaro Videla 是一位在基于 MQ 的应用方面的专业开发者和架构师。他对于 RabbitMQ 的演讲足迹遍布亚洲、欧洲和美国。

Jason J. W. Williams 是 DigiTar 的 CTO。该公司是消息通信服务供应商。从 2008 年开始，他就在公司指导设计和开发工作，包括使用 RabbitMQ 进行实时分析操作。

关于封面插图

本书封面的配图文字是“一位来自克罗地亚科尔丘拉岛上鲁巴达镇的当地农夫”。封面取自 19 世纪中期 Nikola Arsenovic 的一本克罗地亚传统服饰图集的复刻版，于 2003 年发表于克罗地亚斯普利特的人种学博物馆。封面插图是由人种学博物馆一位热心的图书管理员提供的。斯普利特在中世纪时是罗马帝国的核心：大概从公元 304 年起，退位的帝国国王 Diocletian 所居住的皇宫就在这里。这本书中囊括了克罗地亚各个地区的精美彩色插图，并对服饰和日常生活做了介绍。

鲁巴达镇是一个大约有着 1200 名居民的小渔村。它位于克罗地亚西海岸亚得里亚海的科尔丘拉岛东北岸的一个小岛上。封面上的这位农夫穿着该地区典型的工作服。这是一种没有过多修饰的简朴服装（他们通常只在周日和其他特殊场合才会穿着正式服装）。日常工作服由打满补丁的棕色裤子、棕色背心、白色亚麻衬衫和一顶草帽组成。他穿着简单朴素，抽着烟斗，拄着铁锹，在休息的时候等候小白兔自投罗网。

在过去的 200 年中，人们的着装和生活方式都发生了巨大的改变，在当时看来如此丰富的多样性已随时间流逝。现在，来自不同大陆的人都已难以区分，更不用说相隔仅数英里的村子和城镇居民了。也许文化的多样性是我们为了有更多丰富多彩的个人生活而必须付出的代价吧——相应地，人们得到更加多样化和快节奏的高科技生活。

Manning 出版社用两个世纪前各地别具一格的生活方式来诠释计算机行业的诞生与发展，用古老典籍和珍藏册把我们带回那个充满风土人情的年代。

目　录

1 天降奇兵

本章要点

- 开放协议的诉求——AMQP
- RabbitMQ简史
- 安装RabbitMQ
- 第一个程序——Hello World

我们生活在一个实时信息持续可用的世界当中。我们编写的应用程序需要以简单的方式可靠且迅速地路由给众多的接收者。更为重要的是，我们需要找到改变信息接收者的方式，而无须频繁地重写它们。应用程序信息经常会沦为孤岛，新的程序如果不将原始信息的生产者重写（或者推倒重来）的话就无法对其进行访问。你也许会自言自语：“好吧，不过消息队列或者 RabbitMQ 如何帮助我来解决这些问题呢？”让我们先来反思下面的场景为何如此熟悉。

你刚刚为公司的杀手级 Web 应用实现了一个非常棒的认证模块。它看起来非常不错。对于每一个页面单击，程序代码会非常高效地和认证服务器通信以确保用户只能访问他们能够访问的页面。你有点沾沾自喜，因为公司的世界顶级牛油果分布网站上的每一次用户单击都会触发你的代码。正在这时，老板进来和你说公司需要

记录每次成功和失败的权限尝试，以便进行数据挖掘。在你婉转地表达了这应该是认证服务器的工作时，老板没好气地通知你，你无法访问那些数据。认证服务器将数据以适当的格式记录下来。现在这成为你需要解决的问题。仔细思考了目前的情况之后，你的脑袋一阵剧痛。你发现自己不得不去修改认证模块，还有可能在处理的时候中断每个页面。毕竟，这些精彩的代码几乎涉及所有的站点访问。让我们先歇会儿。按下“简单”按钮，时间回到了一开始开发认证模块的那一刻。让我们假设你从第一天开始就决定在设计中好好利用消息队列。

通过使用 RabbitMQ，你聪明地利用消息队列解耦了模块和认证服务器。认证模块被设计为在每一次页面请求时，发送一条认证请求消息到 RabbitMQ。然后认证服务器监听 RabbitMQ 队列并接收该请求消息。一旦请求被获准，认证服务器会向 RabbitMQ 发送一个应答消息。RabbitMQ 会将该消息路由到认证模块所监听的那个队列。在这种模式下，老板的需求不会再让你头疼了。你会发现自己不需要碰模块或者重写一个。所有需要做的是编写一个简单的应用程序连接到 RabbitMQ 并且订阅认证请求。无须更改任何代码。之前编写的代码都不需要知道发生了什么变更。这太简单了，以至于你脸上露出了满意的笑容。这就是消息通信的力量！它让你的日常工作变得如此轻松。

消息队列（message queuing）使用消息将应用程序连接起来。这些消息通过像 RabbitMQ 这样的消息代理服务器在应用程序之间路由。这就像是在应用程序之间放置一个邮局。现实的情况是，这个解决方法并不仅仅针对的是金融行业实时通信问题，它同时也解决了我们开发人员每天要面对的问题。作为作者的我们并没有金融服务行业的背景。当我们需要扩大应用规模的时候，我们搞不清楚什么是“企业消息通信”。我们和你一样只是开发人员，想要解决的这个问题就是处理庞大的实时信息，并把它们快速路由到众多的消费者。我们要在不阻塞消息生产者的情况下做到这一点，同时也无须让生产者知道最终消费者是谁。RabbitMQ 帮助我们轻松解决这些常见问题，并用一种基于标准的方法来确保应用程序之间相互通信，而不管应用是用 Python、PHP 还是 Scala 编写的。

在接下来的几个章节里，我们会带你做更深入的了解。我们会从解释消息队列的工作机制、历史以及为何 RabbitMQ 可以满足需求开始。然后我们会带你漫步于真实世界的案例，你可以将它们应用于自己的可扩展性和可操作性挑战。最后会教你如何将 Rabbit 打造成一部运转良好的机器，“对宕机说再见！”

我们希望本书能指引你迈过消息通信的“荒野”。我们希望你能从我们的经验和伤痕中受益，能够解放自己去创造美妙的应用程序。在结束本章之前，你会清楚地了解消息通信简史，以及 RabbitMQ 的安装和运行。闲话少说，让我们看看消息通信的起源吧。

1.1 住在别人的地下城堡

消息队列的世界并非从潮湿和狭小的地下城开始，其中大多数人屈从于占据该领地的霸主。它起源于另一处拜占庭软件景观的一缕阳光。在 1983 年那年，一位来自孟买的 26 岁工程师脑海里浮现了一个激进的想法：为什么没有一种通用的软件“总线”——一种通信系统，可以解决应用程序间繁重的信息通信工作呢？来自 MIT 的硬件设计教育工作者 Vivek Ranadivé 设想了一种通用软件总线，就像主板上的总线那样，供其他应用程序接入（请查阅 http://hbswk.hbs.edu/archive/1884.html）。因此，在 1983 年 Teknekron 诞生了。手上握着崭新的哈佛 MBA 证书，脑袋里拥有强大的想法，Vivek 为世界各地的开发者开垦了一条捷径。

拥有美好的想法是一回事，而为它找到杀手级的应用则是另一回事。在 1985 年的高盛，Ranadivé 找到了他的第一位客户以及软件总线天生用来解决的问题：金融交易。当时交易员的小隔间内挤满了用来完成交易的不同终端，每台终端上显示着不同类型的信息。Teknekron 发觉这是个机会：替换掉所有那些终端和孤立的应用程序。在交易员位置上的将会是 Ranadivé 的软件总线。而桌上只会剩下一台工作站。而工作站的显示程序可以作为消费者接入 Teknekron 软件总线，并允许交易员“订阅”他想要看到的信息。于是发布订阅模式（PubSub）诞生了，同时还诞生了世界上第一个现代消息队列软件：Teknekron 的 The Information Bus（TIB）。

用不了多久，这种数据传输模型就找到了更多杀手级用途。最终，发布数据的应用和消费数据的应用再也不用直接连接在一起了。它们甚至都不需要对方的存在。Teknekron 的 TIB 允许应用开发者建立一系列规则去描述消息内容。只要消息按照这些规则发布出去，任何消费者应用都能订阅感兴趣的消息。现在，信息的生产者和消费者之间可以完全解耦，并且可以在传输过程中灵活混合。PubSub 模型（生产者 / 消费者）的任意一边可以完全互换而不会影响到另一边。唯一需要保持稳定的是 TIB 软件以及标记和路由信息规则。由于在那个时期金融交易行业门庭若市，因此 TIB 迅速传播开来。它引起了电信特别是新闻机构的注意。这些行业也特别需要

将信息及时发布给千变万化的客户。这也解释了为什么在 1994 年大型新闻机构路透社收购了 Teknekron。

与此同时，这个迅猛发展的企业软件引起了蓝色巨人的注意。毕竟 IBM 最大的几位客户都来自于金融服务行业。而且，Teknekron 的 TIB 软件通常会运行在 IBM 的硬件和操作系统上。除了 White Plains[1] 的那群孩子外所有人都从中获得了好处。因此 20 世纪 80 年代后期，IBM 开始研究开发自己的消息队列软件，运用他们在开发 DB2 时信息投递的丰富经验（参见 http://www-01.ibm.com/software/integration/wmq/MQ15Anniversary.html）。开发工作起始于 1990 年，就在 IBM 的英国温切斯特附近的 Hursely 公园实验室。3 年之后，消息队列服务器软件 IBM MQ 产品系列面世。之后的 17 年，MQ 系列进化成了 WebSphere MQ 并统治着商业消息队列平台市场。在那段时间，Ranadivé 的 TIB 并没有消失在路透社腹中。相反，它仍然是企业通信市场的主要参与者，并且通过更名为 Rendezvous 而走向繁荣。在 1997 年 Teknekron 以 TIBCO 的形式作为一家独立公司再度出现。同年，微软也在消息通信市场崭露头角：微软消息队列（MSMQ）。

通过这一系列的革新，消息队列（message queuing，MQ）软件主要留住了大型组织机构，它们需要可靠性、解耦以及实时消息通信。为什么 MQ 不去寻找更大的市场呢？它是如何度过 20 世纪 90 年代末网络泡沫的呢？毕竟，从 Twitter 到 Salesforce.com，当今所有的这些企业都在努力创建内部方案来解决 25 年前 TIB 就已经解决的 PubSub 问题。一句话：供应商壁垒。商业 MQ 供应商想要解决应用互通的问题，而不是去创建标准接口来允许不同的 MQ 产品互通，或者（但愿不是这样）允许应用程序来更改 MQ 平台。供应商壁垒维持着足够高的价格和利润率，并使得这些商业 MQ 软件对那些当今繁荣昌盛的初创公司和 Web 2.0 公司来说遥不可及。

结果，中小技术公司并不是唯一一个对高价格 MQ 供应商感到不满的。那些造就 MQ 产业的金融服务公司对此也激动不起来。越是大型的金融公司越不可避免地使用来自众多供应商的 MQ 产品，来服务企业内部的不同应用。如果应用已经订阅了 TIBCO MQ 信息，若突然需要消费来自 IBM MQ 的消息，则实现起来会非常困难。这些产品使用不同的 API、不同的协议，因而毫无疑问无法联合起来组成单一的总线。为了解决这个问题，Java Message Service（JMS）在 2001 年诞生了（请查阅 http://en.wikipedia.org/wiki/Java_Message_Service）。JMS 试图通过提供公共 Java

1 IBM总部所在地。——译者注

API 的方式，隐藏单独 MQ 产品供应商提供的实际接口，从而跨越了壁垒和解决了互通问题。从技术上讲，Java 应用程序只需针对 JMS API 编程，选择合适的 MQ 驱动即可。JMS 会打理好其他部分的。问题是你在尝试使用单独标准化接口来胶合众多不同的接口。这就像是把不同类型的衣服黏在一起：缝合处终究会裂开，真相会暴露出来。使用 JMS 的应用程序会变得更加脆弱。我们需要新的消息通信标准化方案。

1.2 救世主AMQP

2004 年，JPMorgan Chase 需要一个更好的消息通信解决方案，并开始和 iMatix 公司一起合作开发 Advanced Message Queuing Protocol（AMQP，高级消息队列协议，参见 http://en.wikipedia.org/wiki/Advanced_Message_Queuing_Protocol#Development）。AMQP 从一开始就设计成为开放标准，以解决众多的消息队列需求和拓扑结构问题。凭借开放，任何人都可以执行这一标准，针对标准编码的任何人都可以和任意 AMQP 供应商提供的 MQ 服务器进行交互。

在很多方面，AMQP 承诺把我们从厂商的“地下城”中解救出来，并且实现 Ranadivé 最初的愿景：从任何发布者到任何感兴趣的消费者之间的信息，通过一条软件总线实时动态连接起来。

1.3 RabbitMQ简史

在本世纪初，一位年轻的伦敦金融部门的企业家创办了一家专门研究 Java 对象缓存的公司：Metalogic。对 Alexis Richardson 来说，理论很简单：使用 Java 对象做分布式计算，同时为了传输性能对它们进行缓存。事实上却大相径庭。不同版本的 Java 虚拟机，以及客户端和服务器端不同的类库，都会使得接收到的对象不可用。Metalogic 方法论的成功之路需要面对现实世界中太多的环境变数。Metalogic 促成了 Alexis 和 Matthias Radestock 的会面（见图 1.1）。

Matthias 在 LShift 工作，而 Alexis 正巧也在那里转租了一件办公室。当时 LShift 正投入和一家大型软件供应商的合同，专注于语言建模和分布式计算。这些领域方面的背景触发了 Matthias 对 Erlang 的兴趣。Erlang 是 Ericsson 原本为了电话交换机而开发的编程语言。引起 Matthias 注意的是 Erlang 在分布式编程和健壮的故

障恢复方面表现出色。可惜的是，当时 Erlang 不是开源的。同时，Metalogic 结束了运营，LShift 的主要分布式计算合同也接近了尾声。Alexis 在 Metalogic 的经历中学到了非常有价值的两课：分布式计算环境的工作机制和哪些公司需要这些环境。

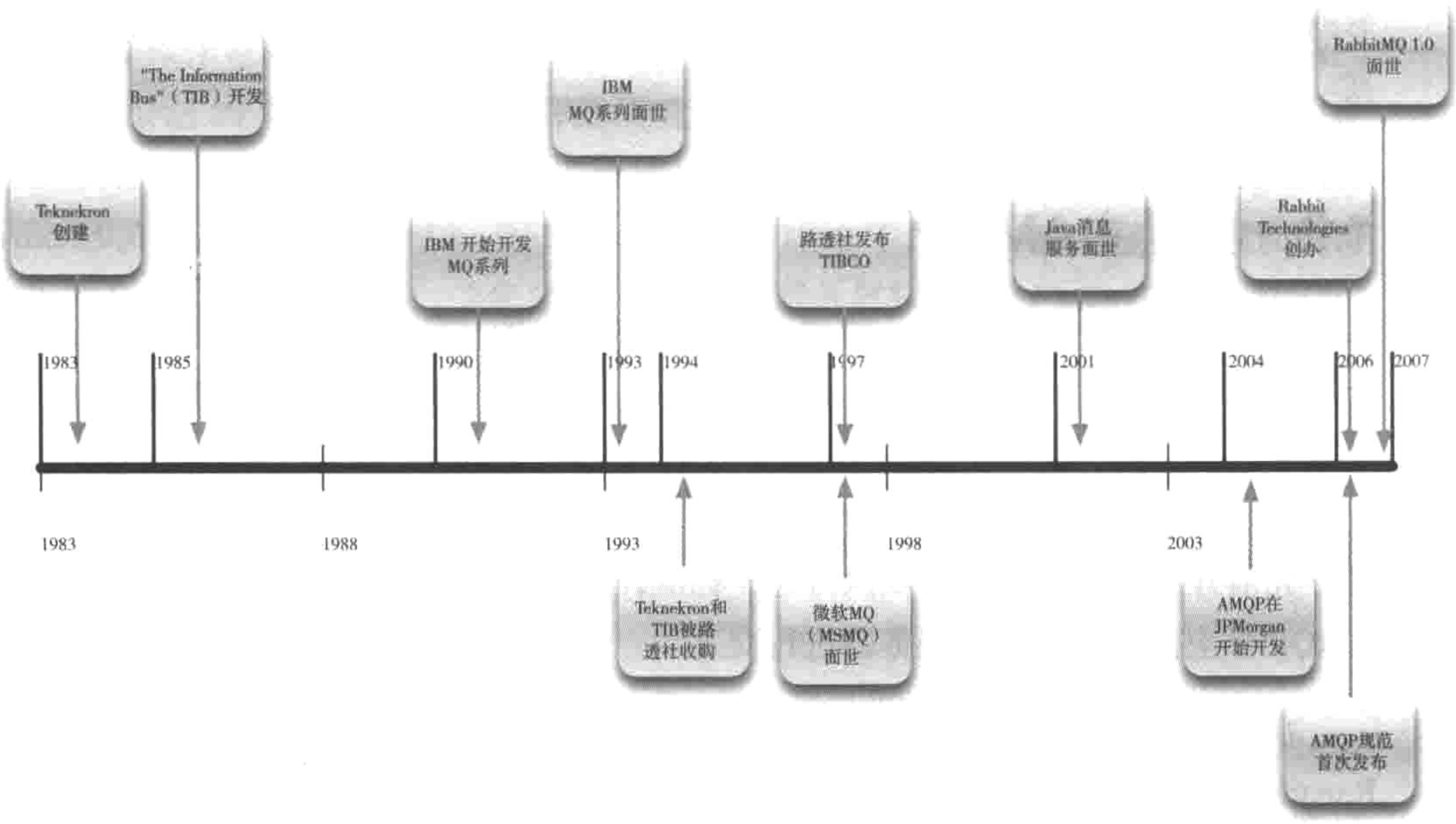

图 1.1 消息队列简史

Alexis 知道自己想要开一家新的公司去解决分布式环境下的通信问题。他也知道他开的下一家公司将会是开源的，并且将采用 JBoss 和 MySQL 成功实践的模型。回顾 Metalogic 解决方案碰到的问题，Alexis 逐渐认识到消息通信才是分布式计算的解决方案。更重要的是，2004 年左右的技术领域正面临开源消息通信的空白。除了商业供应商外，没有人提供消息通信的解决方案。“企业”开源在数据库（MySQL）和应用服务器方面（JBoss）热火朝天，但却没有人去触碰消息通信那片空白。有趣的是：就在 2004 年，在 JPMorgan Chase，AMQP 正开始开发。因为在金融行业的背景，Alexis 被引荐给了在 JPMorgan 的 AMQP 主要驱动者 John O’Hara（将来的 AMQP 工作组创始人）。通过 O’Hara，Alexis 接触到了 AMQP，并开始着手构建 RabbitMQ。

在 2005 年左右，Alexis 创办了 CohesiveFT。他和他的合伙人在美国开办了公司，提供应用栈和工具，在今天逐渐成为云计算。对 Alexis 来说，应用栈的最关键部

分是分布式消息通信。他（仍然在 LShift 那里办公）开始找 Matthias 探讨 AMQP。Matthias 清楚他已经找到了想要用 Erlang 实现的应用。但在动手之前，Alexis 和 Matthias 聚焦在三个问题上。这三个问题是决定了用 Erlang 实现的 AMQP 开源版本能否成功的关键：

（1）那些大型金融机构是否会在意他们的消息代理服务器是用 Erlang 语言编写的？

（2）Erlang 语言是否真的是编写 AMQP 服务器的最佳选择？

（3）如果采用 Erlang 语言编写的话，是否会影响开源社区对它的采纳？

第一个问题立马就被一家金融公司解决了：他们不在乎软件是用什么语言编写的，只要它能够帮助减少集成上的花费即可。第二个问题由就职于 Erlang Solutions 的 Francesco Cesarini 回答：从他对 AMQP 的分析来看，这份规格说明展现的就是每一部电话交换机的架构。换句话说，你无法找到比 Erlang 更好的实现语言来构造 AMQP 代理服务器。最后一个问题则是被完全不同的消息通信服务器 ejabberd 解决了。直到 2005 年，Extensible Messaging and Presence Protocol（XMPP）已经逐渐成为受人尊敬的开放即时通信标准，并且首选的实现就是 Alexey Shchepin 编写的基于 Erlang 的 ejabberd 服务器包。ejabberd 被广泛地使用在众多不同的组织机构里。它是由 Erlang 实现的这一点似乎并未阻挡其流行的步伐。

通过解决这三个主要的问题，Alexis 和 Matthias 说服了 CohesiveFT 和 LShift 共同支持这个项目。他们首先做的是签约了 Matthew Sackman（他现在是 Rabbit 核心开发人员），让他用 Erlang 写了一个原型测试网络延迟。他们很快就发现用 Erlang 编写的分布式计算库有着和原生 socket 一样的延迟，简直不可思议。他们对名称很快有了共识：所有人都认可 Rabbit 这个名字。毕竟，兔子是行动非常迅速的动物而且繁殖起来也非常疯狂，把它用于分布式软件的命名再合适不过了。至少选择 Rabbit 这个名字也便于记忆。因此在 2006 年，Rabbit Technologies 诞生了：一家由 CohesiveFT 和 LShift 的合资企业，其拥有着 RabbitMQ 的知识产权。

时机总是如此的恰到好处，就在当时，AMQP 规范的第一份公开草案也公之于世了。作为一份新的规范，AMQP 正快速修订。这正是 Erlang 可以施展拳脚的地方。通过使用 Erlang，RabbitMQ 可以快速开发并跟上 AMQP 标准前进的节奏。令人惊讶的是，核心开发人员 Tony Garnock-Jones 仅仅用了两个半月的时间就将 RabbitMQ

1.0 版本开发完成了。最初，RabbitMQ 实现了 AMQP 的一个关键特性，使其有别于 TIBCO 和 IBM：使用协议本身就可以对像队列和交换器这样的资源进行配置。对商业供应商来说，资源配置需要通过特定的管理终端的特定工具才可以完成。RabbitMQ 的资源配置能力使其成为构建分布式应用的最完美的通信总线，特别有助于充分利用基于云的资源和快速开发。

就这样一直到今天，RabbitMQ 广泛使用在小到硅谷的初创公司，大到互联网巨头。对 RabbitMQ 来说，那也许是最好的事了，创始人感到非常惊讶：Rabbit 的客户主要是技术公司而并非金融公司。对于那些只有少量预算同时也要解决消息通信的人来说，RabbitMQ 实现了 Ranadivé 的愿景。那正是 RabbitMQ 吸引我们的地方。我们并不知道自己是在找寻消息队列软件。我们知道的是自己需要解决应用集成和高强度事务处理负载的问题。RabbitMQ 为我们提供了一个强大的工具来解决那些问题，并给我们带来了一段丰富的消息通信历史，以及适合所有人的可拔插的信息总线。

1.4 百里挑一

今天，RabbitMQ 并不是开放消息通信的唯一选择。像 ActiveMQ、ZeroMQ 和 Apache Qpid 都提供了不同的开源消息队列方案。问题是，为什么我们认为你应该选择 RabbitMQ 呢？

- 除了 Qpid 之外，RabbitMQ 是唯一实现了 AMQP 标准的代理服务器。
- 正是由于 Erlang，RabbitMQ 集群不可思议的简单。
- 每个人的经历不同，但我们发现 RabbitMQ 比竞争对手更可靠，更能防止崩溃。

也许最重要的原因是 RabbitMQ 难以置信的容易安装和使用。不管是工作站上的单节点设置，还是 Web 基础设施背后的七台服务器集群，RabbitMQ 都能在 30 分钟内安装好并运行起来。记住这些，让我们现在就开始吧。

1.5 在UNIX系统上安装RabbitMQ

到目前为止，我们已经讨论了 AMQP 协议背后的动机以及 RabbitMQ 服务器的历史。现在是时候把代理服务器运行起来开始“倒腾”了。运行 RabbitMQ 的操作

系统需求非常灵活，我们可以在多个平台上运行 RabbitMQ，包括 Linux、Windows、Mac OS X 和其他类 UNIX 系统。

在本章中，我们会介绍在通用 UNIX 系统上安装服务器的流程（本书中所有的示例和指令在未明确指出的情况下，都将假定运行在 UNIX 环境上）。由于 RabbitMQ 是采用 Erlang 编写的，因此我们需要先安装该语言库，以便运行代理服务器。

1.5.1 为什么环境很重要——生活在 Erlang 的世界里

我们建议你使用最新版本的 Erlang。在执笔本书时，最新版是 R14A。你可以从 Erlang 官网上获得 Erlang 的最新发行版（http://www.erlang.org/）。请按照网站上的安装说明进行安装。通过运行最新版本的 Erlang，我们可以获得所有更新和改进。Erlang 每一次新的发布都包含了性能提升。

在解决了 RabbitMQ 的依赖关系后，创建一个文件夹用于测试。假设你运行的是类 UNIX 系统，打开一个终端，输入以下命令：

```
$ mkdir rabbitmqinaction
$ cd rabbitmqinaction
```

1.5.2 获取安装包

然后从官网的服务器下载页面下载 RabbitMQ 服务器：http://www.rabbitmq.com/server.html。选择通用 UNIX 系统平台的安装包，然后进行下载[2]。

```
$ wget http://www.rabbitmq.com/releases/rabbitmq-server/v2.7.0/\
rabbitmq-server-generic-unix-2.7.0.tar.gz
```

下一步你要做的是解压压缩包，并进入 rabbitmq_server-2.7.0 目录：

```
$ tar -xzvf rabbitmq-server-generic-unix-2.7.0.tar.gz
$ cd rabbitmq_server-2.7.0/
```

1.5.3 设置文件夹结构

在启动服务器之前，我们还需要创建一些文件夹。首先第一个是 RabbitMQ 的日志文件夹。如果你需要检查安装情况的话，则可以看看这个文件夹。第二个是

2 你可以从http://www.rabbitmq.com/download.html下载RabbitMQ的预构建安装包，支持Windows、Debian/Ubuntu和RedHat (RPM)。

Mnesia 数据库文件夹。RabbitMQ 使用 Mnesia 数据库存储服务器信息，比如队列元数据、虚拟主机等。在终端输入以下命令：

```
$ mkdir -p /var/log/rabbitmq
$ mkdir -p /var/lib/rabbitmq/mnesia/rabbit
```

如果你以超级用户的身份运行了那些命令的话，记得使用 chown 给系统用户设置权限。

> **小贴士：**在生产环境运行 RabbitMQ 时，我们通常会专门创建一个 rabbitmq 用户，然后为其赋予文件夹权限，而非使用普通用户账户来运行所有命令。

1.5.4 首次运行 Rabbit

现在一切就绪，准备启动服务器。输入以下命令：

```
$ sbin/rabbitmq-server
```

RabbitMQ 会输出一些关于启动过程的信息。如果一切正常，你就会看到 RabbitMQ 的 logo 将以 ASCII 字符的艺术形式展现。消息代理服务器就开始运行了，如图 1.2 所示。

现在打开一个新的终端窗口，并检查服务器的状态。输入以下命令[3]：

```
$ cd path/to/rabbitmqinaction/rabbitmq_server-2.7.0/
$ sbin/rabbitmqctl status
```

3 如果你是从RPM或者Ubuntu/Debian包进行安装的话，就有可能需要以root身份来运行rabbitmqctl。

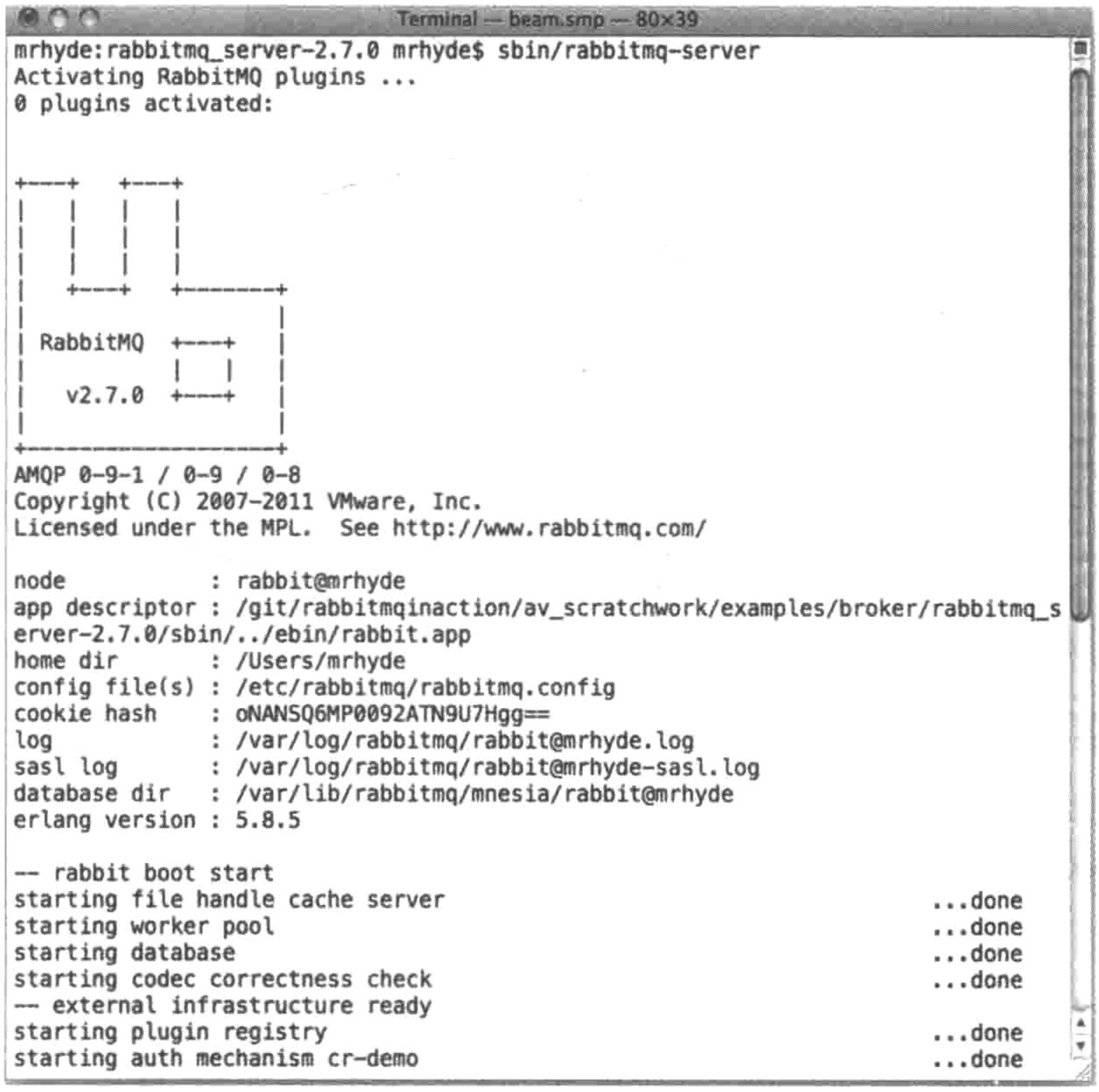

图 1.2 RabbitMQ 的欢迎消息

如图 1.3 所示，这个命令会输出代理服务器的状态、运行中的应用程序和节点。此时此刻，默认配置的 RabbitMQ 代理服务器正运行在你的电脑上。

让我们回顾一下：

- 下载服务器包
- 解压到测试文件夹下
- 设置必需的文件夹结构
- 启动 RabbitMQ 服务器
- 检查服务器状态

通过这些简单的步骤，你就启动了 RabbitMQ。现在我们将介绍更多消息通信理论，然后我们会在代理服务器上运行一些示例。

```
Terminal — bash — 87×28
mrhyde:rabbitmq_server-2.7.0 mrhyde$ sbin/rabbitmqctl status
Status of node rabbit@mrhyde ...
[{pid,7595},
 {running_applications,[{rabbit,"RabbitMQ","2.7.0"},
                        {ssl,"Erlang/OTP SSL application","4.1.6"},
                        {public_key,"Public key infrastructure","0.13"},
                        {crypto,"CRYPTO version 2","2.0.4"},
                        {os_mon,"CPO  CXC 138 46","2.2.7"},
                        {sasl,"SASL  CXC 138 11","2.1.10"},
                        {mnesia,"MNESIA  CXC 138 12","4.5"},
                        {stdlib,"ERTS  CXC 138 10","1.17.5"},
                        {kernel,"ERTS  CXC 138 10","2.14.5"}]},
 {os,{unix,darwin}},
 {erlang_version,"Erlang R14B04 (erts-5.8.5) [source] [64-bit] [smp:2:2] [rq:2] [async-
threads:30] [hipe] [kernel-poll:true]\n"},
 {memory,[{total,25627552},
          {processes,10422296},
          {processes_used,10407064},
          {system,15205256},
          {atom,1139097},
          {atom_used,1137479},
          {binary,76104},
          {code,11346167},
          {ets,901984}]},
 {vm_memory_high_watermark,0.3999999999441615},
 {vm_memory_limit,2865407590}]
...done.
mrhyde:rabbitmq_server-2.7.0 mrhyde$
```

图 1.3 检测 RabbitMQ 状态

1.6 总结

你现在能明白我们为何如此喜欢 RabbitMQ 了吧。即便作为来自金融行业的技术后代，RabbitMQ 设置起来仍很方便。那些曾由 TIBCO 和 IBM 引领创作的复杂路由和可靠性特性现在都为你所用。最激动人心的是，它是开源的！我们已经向你展示了过去 30 年来消息通信的由来，从将金融交易系统连接起来的软件总线，到成为金融交换器“心脏”的消息路由怪兽，再到半导体晶圆厂的生产线。我们才刚刚完成第 1 章而已，你就能在开发笔记本电脑上将它运行起来了！接下来该研究消息通信的各个模块了，包括队列、绑定、交换器以及虚拟主机。让我们看看如何将它们组装在一起，并在 Rabbit 上运行“Hello World”！

2 理解消息通信

本章要点

- 消息通信概念——消费者、生产者和代理
- AMQP元素——交换器、队列和绑定
- 虚拟主机
- 消息持久化
- 一条消息历经从生产者到消费者的生命周期

当提到消息通信（messaging）时，程序员们会联想到许多不同的事物。脑海里最先浮现的是邮箱和IM（即时通信），不过这些模型并非我们讨论的RabbitMQ消息通信。虽然在某些方面，RabbitMQ的消息通信和邮箱、IM有共同之处，但它们却是完全不同的模式。举例来说，虽然AMQP（高级消息队列协议）像邮箱那样为离线消费者存储消息，但是这些根据标签路由的消息更为灵活。同时和邮件不同的是，这些消息没有固定的结构，甚至可以直接存储二进制数据。不同于IM协议，AMQP隐去了消息的发送方和接收方。AMQP也没有“存在”[1]这个概念。因此，这

1 主流的IM协议有四种，即时消息和存在协议（Instant Messaging and Presence Protocol，IMPP）、存在和即时消息协议（Presence and Instant Messaging Protocol，PRIM）、SIP即时消息和存在扩展协议（SIP for Instant Messaging and Presence Leveraging Extensions，SIMPLE）和可扩展消息与存在协议（Extensible Messaging and Presence Protocol，XMPP）。——译者注

个灵活的基础设施鼓励你解耦应用程序之间的通信。AMQP 消息能以一对多的广播方式进行路由，也可以选择以一对一的方式路由。在 IM 中，你只能一对一通信。

由于 AMQP 消息通信与其他通信协议不同，因此我们将会在接下来的几节中解释 AMQP 中的术语和构造。如果你有像 TIBCO 或者 IBM MQ 系列这样的企业级消息通信系统的良好基础，那你会觉得很多内容其实非常熟悉。因为 RabbitMQ 专注于应用程序之间的消息通信，所以清楚地理解通信模式的概念是非常重要的。让我们忘记根植于脑海的客户端 / 服务器端特性，从现在开始熟悉消费者和生产者的概念吧。

2.1 消费者和生产者（这可不是经济学课程哦）

如果你曾在工作中使用过网络软件，脑海中应该会有客户端和服务器端的概念。不管是浏览器和 Web 服务器，还是应用程序和 MySQL 服务器，都是其中一方发送请求，而另一方服务这些请求。你可以将其视为快餐车模式。你的应用程序下订单，然后快餐车完成订单。你需要的数据来源于快餐车服务器。该模型就是我们平时如何尝试理解应用程序和服务器之间发生的一切。因此对于这个新的消息通信机制，你可能会问：哪个是顾客，哪个是快餐车，还有我怎样下订单呢？

这确实是个问题。RabbitMQ 不是快餐车而是消息投递服务。应用程序从 RabbitMQ 获得的数据并不是由 Rabbit 产生的，就如同你收到的快递包裹也不是由 FedEx 生产的一样。因此，让我们把 Rabbit 当作一种投递服务。应用程序可以发送和接收包裹。而数据所在的服务器也可以发送和接收。RabbitMQ 在应用程序和服务器之间扮演着路由器的角色。所以当应用程序连接到 RabbitMQ 时，它就必须做个决定：我是在发送还是在接收呢？或者从 AMQP 的角度思考，我是一个生产者还是一个消费者呢？

生产者（producer）创建消息，然后发布（发送）到代理服务器（RabbitMQ）。什么是消息呢？消息包含两部分内容：有效载荷（payload）和标签（label）。有效载荷就是你想要传输的数据。它可以是任何内容，一个 JSON 数组或者是你最喜欢的 iguana Ziggy 的 MPEG-4。RabbitMQ 不会在意这些。标签就更有趣了。它描述了有效载荷，并且 RabbitMQ 用它来决定谁将获得消息的拷贝。举例来说，不同于 TCP 协议的是，当你明确指定发送方和接收方时，AMQP 只会用标签表述这条消息

（一个交换器的名称和可选的主题标记），然后把消息交由 Rabbit。Rabbit 会根据标签把消息发送给感兴趣的接收方。这种通信方式是一种“发后即忘”（fire-and-forget）的单向方式。稍后在我们讨论交换器和绑定时，会更详细地探讨 RabbitMQ 是如何解释标签的。眼下你只需要知道生产者会创建消息并设置标签（见图 2.1）。

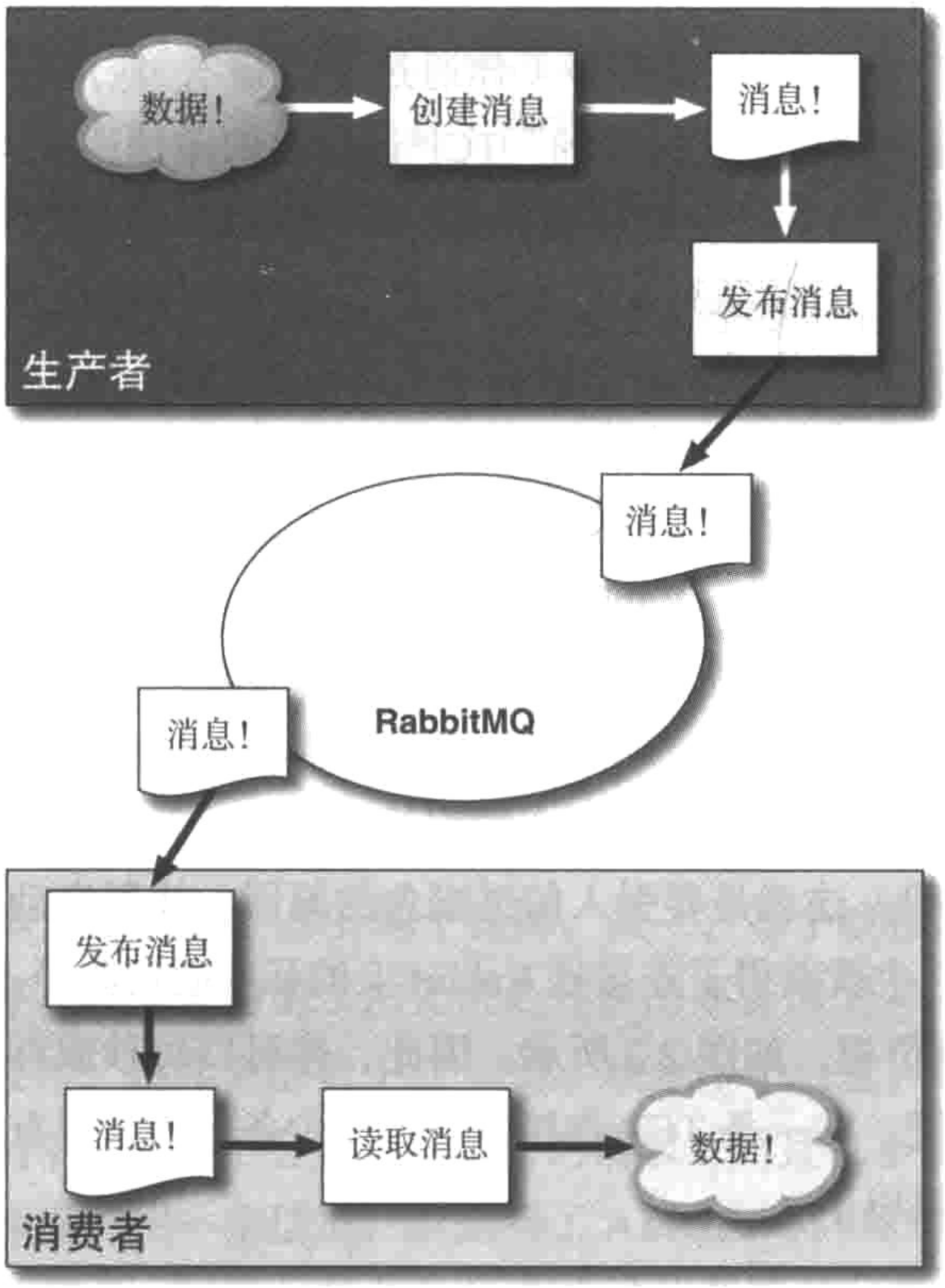

图 2.1 从生产者到消费者的消息流

消费者很容易理解。它们连接到代理服务器上，并订阅到队列（queue）上。把消息队列想象成一个具名邮箱。每当消息到达特定的邮箱时，RabbitMQ 会将其发送给其中一个订阅的 / 监听的消费者。当消费者接收到消息时，它只得到消息的一部分：有效载荷。在消息路由过程中，消息的标签并没有随有效载荷一同传递。RabbitMQ 甚至不会告诉你是谁生产 / 发送了消息。就好比你拿起信件时，却发现所有的信封都是空白的。想要知道这条消息是否是从 Millie 姑妈发来的唯一方式是她在信里签了名。同理，如果需要明确知道是谁生产的 AMQP 消息的话，就要看生产者是否把发送方信息放入有效载荷中。

整个过程其实很简单：生产者创建消息，消费者接收这些消息。你的应用程序可以作为生产者，向其他应用程序发送消息。或者作为一个消费者，接收消息。也可以在两者之间进行切换。不过在此之前，它必须先建立一条信道(channel)。等等！什么是信道呢?

你必须首先连接到 Rabbit，才能消费或者发布消息。你在应用程序和 Rabbit 代理服务器之间创建一条 TCP 连接。一旦 TCP 连接打开（你通过了认证），应用程序就可以创建一条 AMQP 信道。信道是建立在“真实的”TCP 连接内的虚拟连接。AMQP 命令都是通过信道发送出去的。每条信道都会被指派一个唯一 ID（AMQP 库会帮你记住 ID 的）。不论是发布消息、订阅队列或是接收消息，这些动作都是通过信道完成的。你也许会问为什么我们需要信道呢？为什么不直接通过 TCP 连接发送 AMQP 命令呢？主要原因在于对操作系统来说建立和销毁 TCP 会话是非常昂贵的开销。假设应用程序从队列消费消息，并根据服务需求合理调度线程。假设你只进行 TCP 连接，那么每个线程都需要自行连接到 Rabbit。也就是说高峰期有每秒成百上千条连接。这不仅造成 TCP 连接的巨大浪费，而且操作系统每秒也就只能建立这点数量的连接。因此，你可能很快就碰到性能瓶颈了。如果我们为所有线程只使用一条 TCP 连接以满足性能方面的要求，但又能确保每个线程的私密性，就像拥有独立连接一样的话，那不就非常完美吗？这就是要引入信道概念的原因。线程启动后，会在现成的连接上创建一条信道，也就获得了连接到 Rabbit 上的私密通信路径，而不会给操作系统的 TCP 栈造成额外负担，如图 2.2 所示。因此，你可以每秒成百上千次地创建信道而不会影响操作系统。在一条 TCP 连接上创建多少条信道是没有限制的。把它想象成一束光纤电缆就可以了。

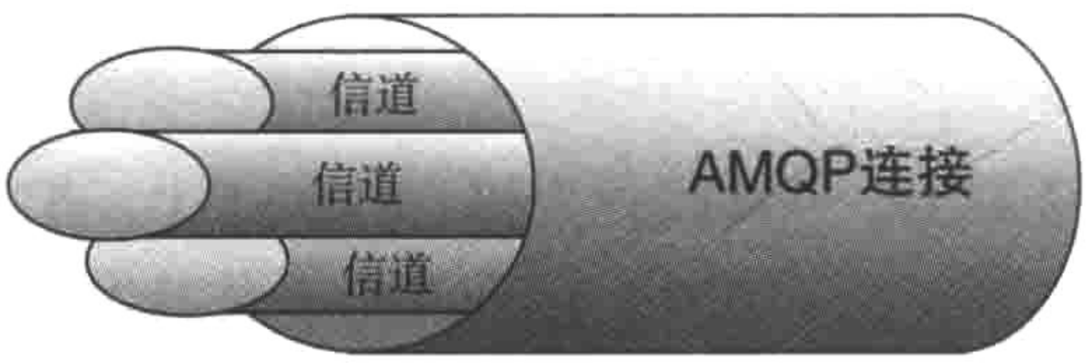

图 2.2 理解 AMQP 信道和连接

每条电缆中的光纤束都可以传输（就像一条信道）。一条电缆有许多光纤束，允许所有连接的线程通过多条光纤束同时进行传输和接收。TCP 连接就像电缆，而 AMQP 信道就像一条条独立光纤束。

我们来举个例子吧。假设你正在编写一个服务用来跟踪代客泊车。所有人都和

RabbitMQ 进行交流。服务必须要完成两个任务：

（1）存储代客票 ID 以及对应的车辆停放泊车位。

（2）返回指定代客票 ID 对应的泊车位。

就第一个任务而言，你提供的服务将扮演消费者的角色。它订阅 Rabbit 队列，等待“存放票”消息。该消息包含票 ID 和泊车位号码。对于第二个任务，你提供的服务既是消费者也是生产者。它需要接收消息来获取特定代客票 ID，然后它需要发布一个包含对应泊车位号码的应答消息。

为了实现第二个任务，应用程序要扮演生产者的角色。一旦建立到 RabbitMQ 代理服务器的连接，应用程序将创建多条信道：chan_recv 信道用于服务接收消息的线程，chan_sendX（X 就是线程号）信道用于服务每一个应答线程。你使用 chan_recv 设置队列的订阅，用来接收包含“票查询”请求的消息。当应用程序通过 chan_recv 信道收到一条票查询消息时，它检查消息中包含的票 ID。一旦确认了对应的泊车位号，应用程序将创建线程发送应答（原始线程则继续接收新的请求）。然后新的应答线程创建包含泊车位号的消息。最终，新线程为应答消息设置标签并通过 chan_sendX 信道将其发送给 Rabbit。如果只有一条信道，新应答线程将无法分享 TCP 连接。你有两个选择。其一，每个线程使用一个连接。这意味着你的应用程序在响应当前请求前无法处理新的票查询请求。其二，为每个发送线程都分配 TCP 连接，这样会浪费 TCP 资源。使用多个信道，线程可以同时共享连接。这意味着对请求的应答不会阻塞消费新的请求，而且也不会浪费 TCP 连接。有时，你可能会选择仅使用一条信道，但是有了 AMQP，你可以灵活地使用多个信道来满足应用程序的需求，而不会有众多 TCP 连接的开销。

重要的是记住消费者和生产者是消息发送和消息接收概念的体现，而非客服端和服务器端。从总体上来说，消息通信，特别是 AMQP，可以被当作加强版的传输层。使用信道，你能够根据应用需要，尽可能多地创建并行的传输层，而不会被 TCP 连接约束所限制。当你理解了这些概念时，你就能把 RabbitMQ 看作软件的路由器了。

2.2 从底部开始构造：队列

你已对消费者和生产者有了一定的了解，已经急着要开始了吧？先别急。首先你需要理解什么是队列。从概念上来讲，AMQP 消息路由必须有三部分：交换器、

队列和绑定。生产者把消息发布到交换器上；消息最终到达队列，并被消费者接收；绑定决定了消息如何从路由器路由到特定的队列。在你研究交换器和绑定之前，需要先理解队列的概念和工作原理。看一下图 2.3。

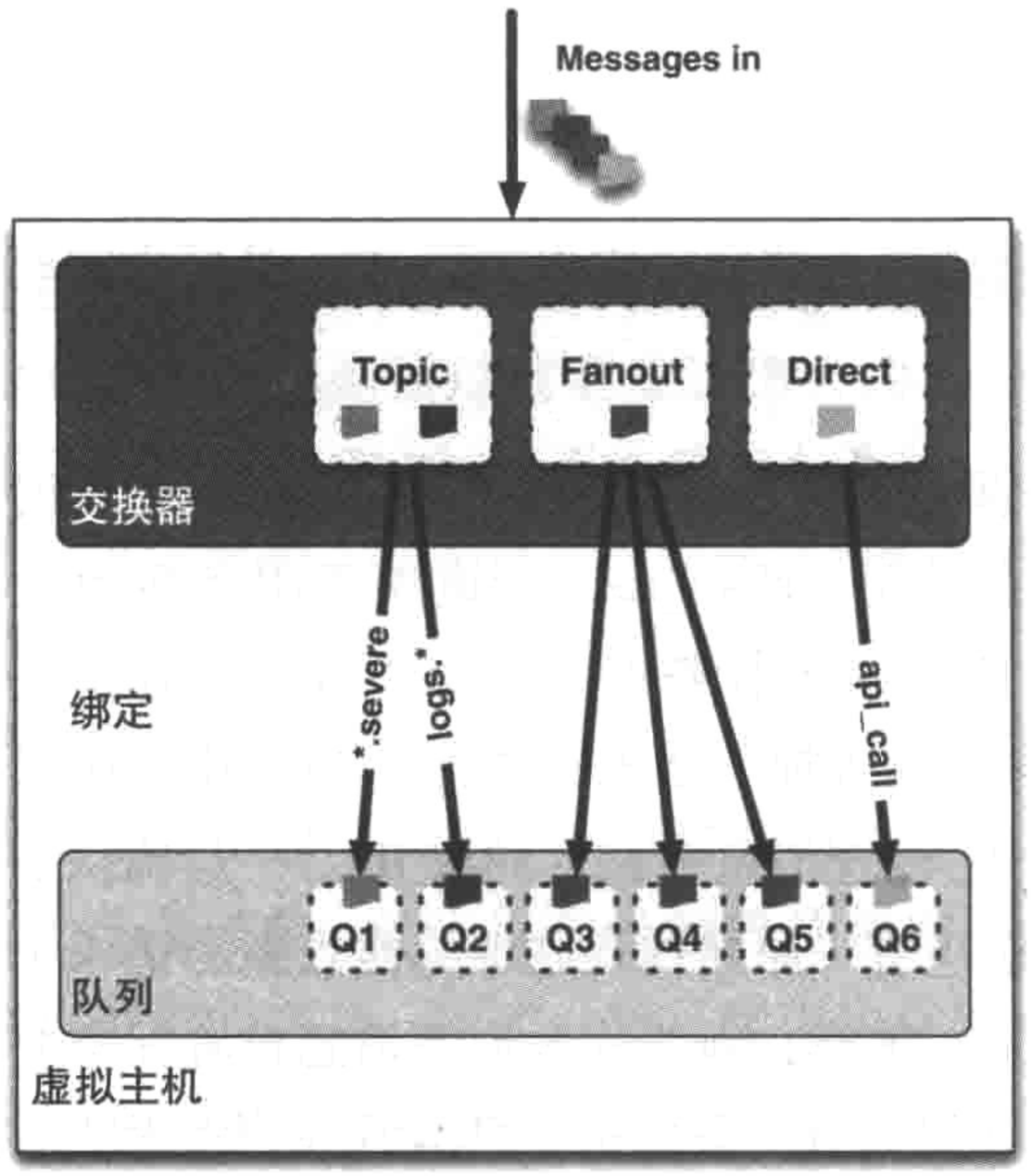

图 2.3 AMQP 栈：交换器、绑定，以及队列

就像我们之前在讨论生产者和消费者时说的那样，队列就如同具名邮箱。消息最终达到队列中并等待消费。消费者通过以下两种方式从特定的队列中接收消息：

（1）通过 AMQP 的 basic.consume 命令订阅。这样做会将信道置为接收模式，直到取消对队列的订阅为止。订阅了消息后，消费者在消费（或者拒绝）最近接收的那条消息后，就能从队列中（可用的）自动接收下一条消息。如果消费者处理队列消息，并且 / 或者需要在消息一到达队列时就自动接收的话，你应该使用 basic.consume。

（2）某些时候，你只想从队列获得单条消息而不是持续订阅。向队列请求单条消息是通过 AMQP 的 basic.get 命令实现的。这样做可以让消费者接收队列中的下一条消息。如果要获得更多消息的话，需要再次发送 basic.get 命令。你不应该将 basic.get 放在一个循环里来替代 basic.consume。因为这样做会影响 Rabbit 的性能。大致上讲，basic.get 命令会订阅消息，获得单条消息，然后取消订阅。消费者理应始终使用 basic.consume 来实现高吞吐量。

如果至少有一个消费者订阅了队列的话，消息会立即发送给这些订阅的消费者。但是如果消息到达了无人订阅的队列呢？在这种情况下，消息会在队列中等待。一旦有消费者订阅到该队列，那么队列上的消息就会发送给消费者。更有趣的问题是，当有多个消费者订阅到同一队列上时，消息是如何分发的。

当 Rabbit 队列拥有多个消费者时，队列收到的消息将以循环（round-robin）的方式发送给消费者。每条消息只会发送给一个订阅的消费者。假设有 seed_bin 队列，消费者 Farmer Bob 和消费者 Farmer Esmeralda 订阅到 seed_bin 队列。 当消息到达 seed_bin 队列时，消息投递方式如下：

（1）消息 Message_A 到达 seed_bin 队列。

（2）RabbitMQ 把消息 Message_A 发送给 Farmer Bob。

（3）Farmer Bob 确认接收到了消息 Message_A。

（4）RabbitMQ 把消息 Message_A 从 seed_bin 中删除。

（5）消息 Message_B 到达 seed_bin 队列。

（6）RabbitMQ 把消息 Message_B 发送给 Farmer Esmeralda。

（7）Farmer Esmeralda 确认接收到了消息 Message_B。

（8）RabbitMQ 把消息 Message_B 从 seed_bin 中删除。

你可能注意到了 Farmers Bob 和 Esmeralda 做了一些我们还未讨论过的事情：他们对消息进行了确认。消费者接收到的每一条消息都必须进行确认。消费者必须通过 AMQP 的 `basic.ack` 命令显式地向 RabbitMQ 发送一个确认，或者在订阅到队列的时候就将 `auto_ack` 参数设置为 true。当设置了 `auto_ack` 时，一旦消费者接收消息，RabbitMQ 会自动视其确认了消息。需要记住的是，消费者对消息的确认和告诉生产者消息已经被接收了这两件事毫不相关。因此，消费者通过确认命令告诉 RabbitMQ 它已经正确地接收了消息，同时 RabbitMQ 才能安全地把消息从队列中删除。

如果消费者收到一条消息，然后确认之前从 Rabbit 断开连接（或者从队列上取消订阅），RabbitMQ 会认为这条消息没有分发，然后重新分发给下一个订阅的消费者。如果你的应用程序崩溃了，这样做可以确保消息会被发送给另一个消费者进行处理。

另一方面，如果应用程序有 bug 而忘记确认消息的话，Rabbit 将不会给该消费者发送更多消息了。这是因为在上一条消息被确认之前，Rabbit 会认为这个消费者并没有准备好接收下一条消息。你可以好好利用这一点。如果处理消息内容非常耗时，则你的应用程序可以延迟确认该消息，直到消息处理完成。这样可以防止 Rabbit 持续不断的消息涌向你的应用而导致过载。

在收到消息后，如果你想要明确拒绝而不是确认收到该消息的话，该如何呢？举例来说，假设在处理消息的时候你遇到了不可恢复的错误，但是由于硬件问题，只影响到当前的消费者（这就是一个很好的示例，直到消息处理完成之前，你绝不能进行确认）。只要消息尚未确认，则你有以下两个选择：

（1）把消费者从 RabbitMQ 服务器断开连接。这会导致 RabbitMQ 自动重新把消息入队并发送给另一个消费者。这样做的好处是所有的 RabbitMQ 版本都支持。缺点是，这样连接 / 断开连接的方式会额外增加 RabbitMQ 的负担（如果消费者在处理每条消息时都遇到错误的话，会导致潜在的重大负荷）。

（2）如果你正使用 RabbitMQ 2.0.0 或者更新的版本，那就使用 AMQP 的 `basic.reject` 命令。顾名思义：`basic.reject` 允许消费者拒绝 RabbitMQ 发送的消息。如果把 `reject` 命令的 `requeue` 参数设置成 true 的话，RabbitMQ 会将消息重新发送给下一个订阅的消费者。如果设置成 false 的话，RabbitMQ 立即会把消息从队列中移除，而不会把它发送给新的消费者。你也可以通过对消息确认的方式来简单地忽略该消息（这种忽略消息的方式的优势在于所有版本的 RabbitMQ 都支持）。如果你检测到一条格式错误的消息而任何一个消费者都无法处理的时候，这样做就十分有用。

> **注意：**当丢弃一条消息时，为什么要使用 `basic.reject` 命令，并将 `requeue` 参数设置成 false 来替代确认消息呢？在将来的 RabbitMQ 版本中会支持一个特殊的“死信”（dead letter）队列，用来存放那些被拒绝而不重入队列的消息。死信队列让你通过检测拒绝 / 未送达的消息来发现问题。如果应用程序想自动从死信队列功能中获益的话，需要使用 `reject` 命令并将 `requeue` 参数设置成 false。

还有一件更重要的事情：如何创建队列。消费者和生产者都能使用 AMQP 的 `queue.declare` 命令来创建队列。但是如果消费者在同一条信道上订阅了另一个队列的话，就无法再声明队列了。必须首先取消订阅，将信道置为“传输”模式。

当创建队列时，你常常想要指定队列名称。消费者订阅队列时需要队列名称，并在创建绑定时也需要指定队列名称。如果不指定队列名称的话，Rabbit 会分配一个随机名称并在 queue.declare 命令的响应中返回（对于构建在 AMQP 上的 RPC 应用来说，使用临时“匿名”队列很有用，我们会在第 4 章学习它）。以下是队列设置中另一些有用的参数：

- exclusive——如果设置为 true 的话，队列将变成私有的，此时只有你的应用程序才能够消费队列消息。当你想要限制一个队列只有一个消费者的时候很有帮助。
- auto-delete——当最后一个消费者取消订阅的时候，队列就会自动移除。如果你需要临时队列只为一个消费者服务的话，请结合使用 auto-delete 和 exclusive。当消费者断开连接时，队列就被移除了。

如果尝试声明一个已经存在的队列会发生什么呢？只要声明参数完全匹配现存的队列的话，Rabbit 就什么都不做，并成功返回，就好像这个队列已经创建成功一样（如果参数不匹配的话，队列声明尝试会失败）。如果你只是想检测队列是否存在，则可以设置 queue.declare 的 passive 选项为 true。在该设置下，如果队列存在，那么 queue.declare 命令会成功返回；如果队列不存在的话，queue.declare 命令不会创建队列而会返回一个错误。

当设计应用程序时，你最有可能会问自己，是该由生产者还是消费者来创建所需的队列呢？看起来最自然的答案是由消费者来创建队列。毕竟，消费者才需要订阅队列，而且总不能订阅一个不存在的队列，是吧？先别这么快下结论。你首先需要想清楚消息的生产者能否承担得起丢失消息。发送出去的消息如果路由到了不存在的队列的话，Rabbit 会忽略它们。因此，如果你不能承担得起消息进入“黑洞”而丢失的话，你的生产者和消费者就都应该尝试去创建队列。另一方面，如果你能承担得起丢失消息，或者你实现一种方法来重新发布未处理的消息的话（我们会向你展现如何做到这一点），你可以只让自己的消费者来声明队列。

队列是 AMQP 消息通信的基础模块：

- 为消息提供了处所，消息在此等待消费。
- 对负载均衡来说，队列是绝佳方案。只需附加一堆消费者，并让 RabbitMQ 以循环的方式均匀地分配发来的消息。

- 队列是 Rabbit 中消息的最后的终点（除非消息进入了“黑洞”）。

在掌握了队列之后，你已经准备好进入下一个 Rabbit 构造块：交换器和绑定！

2.3 联合起来：交换器和绑定

就像在前几节看到的那样，你想让消费者从队列中获取消息。现在的问题是，消息是如何到达队列的呢？让我们来认识一下 AMQP 的交换器和绑定。当你想要将消息投递到队列时，你通过把消息发送给交换器来完成。然后，根据确定的规则，RabbitMQ 将会决定消息该投递到哪个队列。这些规则被称作路由键（routing key）。队列通过路由键绑定到交换器。当你把消息发送到代理服务器时，消息将拥有一个路由键——即便是空的——RabbitMQ 也会将其和绑定使用的路由键进行匹配。如果相匹配的话，那么消息将会投递到该队列。如果路由的消息不匹配任何绑定模式的话，消息将进入“黑洞”。

为何要如此大费周章？你也许会这么说：“我只想让消息到达队列而已。”让我们看一个例子，以便理解上述概念的优势。

你可以将这个场景和邮件进行比较：如果你想把一条消息发送给任何一个联系人，则只需把消息发送到对方的邮件地址，SMTP 服务器会检查消息是发送给谁的并会负责投递到用户的收件箱。但是，如果你的联系人想要把来自你的每条消息都归档到商务文件夹下的话会如何呢？为了能达成这个目标，他们需要根据消息内容设置明确的规则。举例来说，他们可能也想通过设置基于主机名的规则，将某些商业供应商归类到同一个文件夹中。通过交换器、绑定和队列的概念，AMQP 实现上述以及更多的使用场景，因此你能将队列绑定到交换器上，而不使用路由键，然后你发送给交换器的每一条没有路由键的消息会投递到上述的队列中去。这一点和邮件系统非常类似。如果你需要复杂的使用案例，例如发布 / 订阅或者多播，则也能够轻易完成。一会儿你就能看到。

除了可以用交换器和绑定来完成不同的使用场景之外，还有另一个好处是：对于发送消息给服务器的发布者来说，它不需要关心服务器的另一端（整个消息处理环节中的队列和消费者）的逻辑。就如你将看到的那样，这可以导致有趣的消息通信场景。如果服务器只允许你直接发布消息到队列的话，是无法实现这些场景的——或者说非常难以实现。

就像你之前看到的那样，服务器会根据路由键将消息从交换器路由到队列，但它是如何处理投递到多个队列的情况的呢？协议中定义的不同类型交换器发挥了作用。一共有四种类型：direct、fanout、topic 和 headers。每一种类型实现了不同的路由算法。我们会讲解除了 headers 交换器外的其他三种。headers 交换器允许你匹配 AMQP 消息的 header 而非路由键。除此之外，headers 交换器和 direct 交换器完全一致，但性能会差很多。因此它并不太实用，而且几乎再也用不到了。让我们仔细看看其他几种类型的交换器吧。

direct 交换器非常简单：如果路由键匹配的话，消息就被投递到对应的队列。请参考图 2.4。

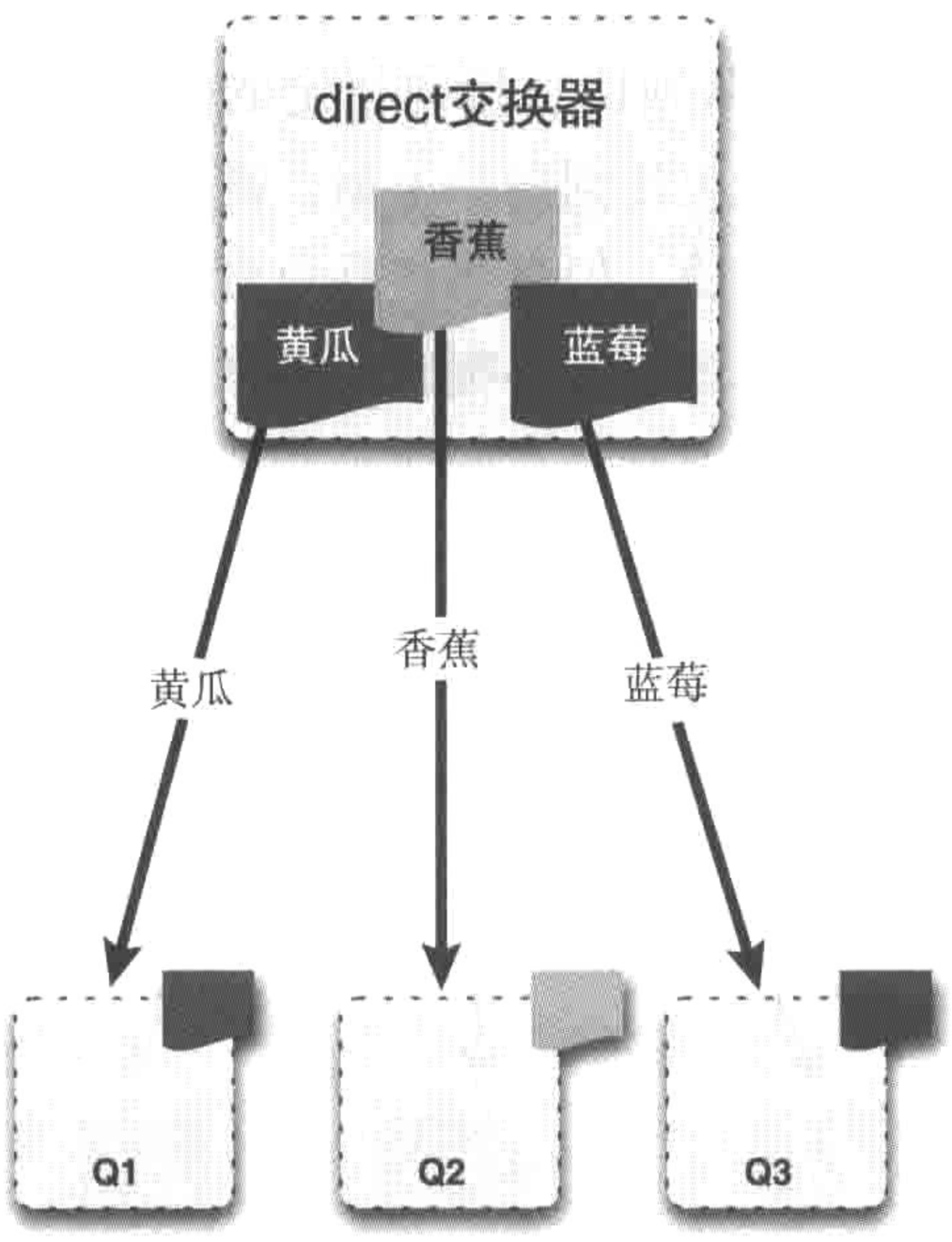

图 2.4 direct 交换器消息流

服务器必须实现 direct 类型交换器，包含一个空白字符串名称的默认交换器。当声明一个队列时，它会自动绑定到默认交换器，并以队列名称作为路由键。这意味着你可以使用如下代码发送消息到之前声明的队列去。前提是你已经获得了信道实例：

```
$channel->basic_publish($msg, '' , 'queue-name' );
```

第一个参数是你想要发送的消息内容；第二个参数是一个空的字符串，指定了

默认交换器；而第三个参数就是路由键了。这个路由键就是之前用来声明队列的名称。之后，你会看到如何使用默认交换器和临时队列来实现 RPC 消息通信模式。

当默认的 direct 交换器无法满足应用程序的需求时，你可以声明你自己的交换器。只需发送 `exchange.declare` 命令并设置合适的参数就行了。

接下来我们要讨论的是 fanout 交换器。就如你看到的图 2.5 那样，这种类型的交换器会将收到的消息广播到绑定的队列上。消息通信模式很简单：当你发送一条消息到 fanout 交换器时，它会把消息投递给所有附加在此交换器上的队列。这允许你对单条消息做不同方式的反应。举例来说，一个 Web 应用程序可能需要在用户上传新的图片时，用户相册必须清除缓存，同时用户应该得到些积分奖励。你可以将两个队列绑定到图片上传交换器上。一个用于清除缓存，另一个用于增加用户积分。从这个场景中你可以了解到，使用交换器、绑定和队列比直接向指定的队列发送消息要有优势。假设应用程序的第一个需求是在图片上传到网站上后，需要清除用户相册缓存。你可以通过只使用一个队列就能轻易完成。但是当产品负责人让你实现一个新功能，即在上传完成后给用户一点奖励，你该怎么办呢？如果你是直接将消息发送给队列的话，就不得不修改发送方的代码，以将消息发送给新的用户积分（points）队列。如果你使用的是 fanout 交换器的话，你唯一需要做的就是为新的消

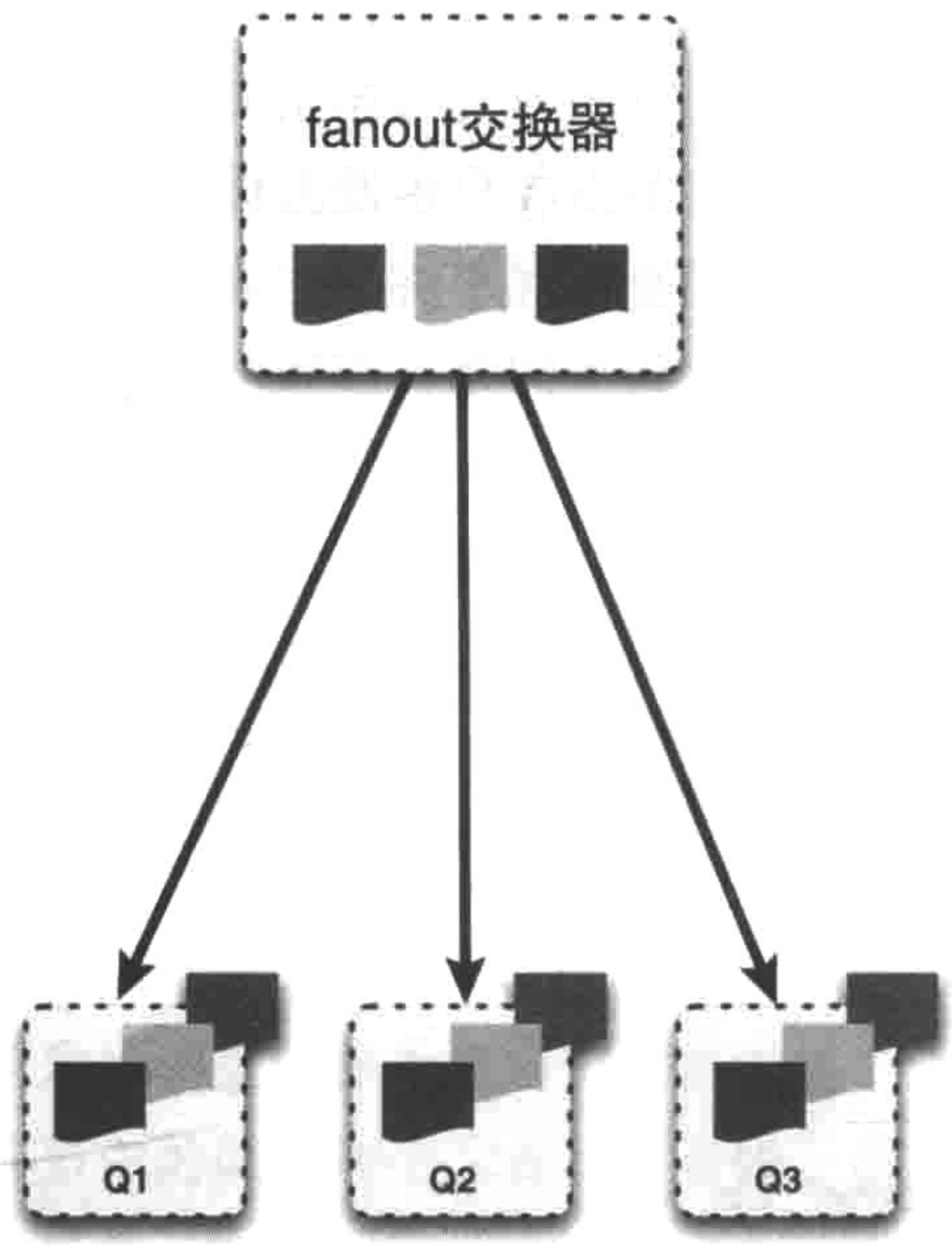

图 2.5 fanout 交换器消息流

费者写一段代码，然后声明新的队列并将其绑定到 fanout 交换器上。就如同我们之前讲的那样，发送方的代码和消费者的代码两者之间完全解耦了，这允许你轻而易举地添加应用程序的功能。

最后，我们将讨论 topic 交换器。这类交换器允许你实现有趣的消息通信场景，它使得来自不同源头的消息能够到达同一个队列。让我们用 Web 应用程序日志系统作为示例。你拥有多个不同的日志级别，例如 error、info 和 warning。与此同时，你的应用程序分为以下几个模块：user-profile、image-gallery、msg-inbox 等。参见图 2.6，如果在发送消息的动作失败时，你想要报告一个 error 的话，则可以编写以下代码：

```
$channel->basic_publish($msg, 'logs-exchange' , 'error.msg-inbox' );
```

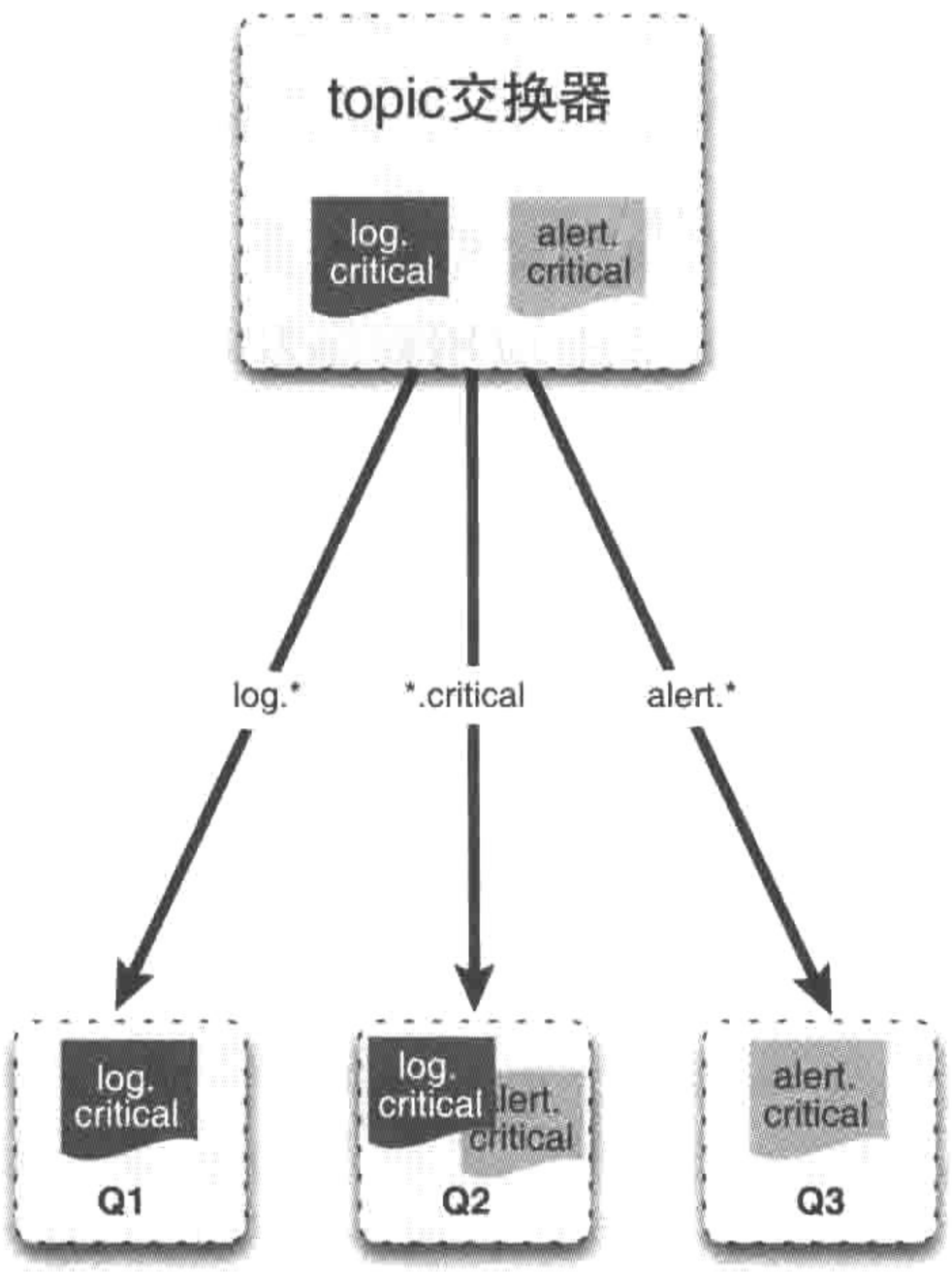

图 2.6 topic 交换器消息流

然后，假设你声明了一个 msg-inbox-errors 队列，你可以将其绑定到交换器上来接收消息，如下所示：

```
$channel->queue_bind('msg-inbox-errors',
                     'logs-exchange',
                     'error.msg-inbox');
```

到目前为止，这看起来和使用 direct 交换器很像。你为队列绑定操作和消息发布路由键指定了相同的 error.msg-inbox 字符串作为绑定规则。那样就能确保你的消息会路由到 msg-inbox-errors 队列，这没什么特别的。但是如果你想要一个队列监听 msg-inbox 模块的所有 error 级别的话，你该怎么做呢？你可以通过将新的队列绑定到已有的同一个交换器来实现，就像下面这样：

```
$channel->queue_bind('msg-inbox-logs',
                     'logs-exchange',
                     '*.msg-inbox');
```

msg-inbox-logs 队列将会接收从 msg-inbox 模块发来的所有 error、warning 和 info 日志信息。那么如何接收所有的日志呢？这实现起来也很简单。你可以在队列绑定到交换器上的时候使用通配符。从之前的例子可以看到，单个“.”把路由键分为了几部分，“*”匹配特定位置的任意文本。为了实现匹配所有规则，你可以使用“#”字符：

```
$channel->queue_bind('all-logs', 'logs-exchange', '#');
```

通过这样的绑定方式，all-logs 队列将会接收所有从 Web 应用程序发布的日志。当然，要使得之前的示例能够运行，你必须在绑定之前对队列进行声明。“*”操作符将“.”视为分隔符；与之不同的是，“#”操作符没有分块的概念，它将任意“.”字符均视为关键字的匹配部分。

到目前为止，你应该已经理解了这三种交换器类型，并能体会 AMQP 的强大之处了。你可以对服务器的行为编程以满足自己的需求。它既能够以发布 / 订阅模式的设置方式作为队列服务器使用，也可以作为 RPC 服务器。这取决于你如何组织这些功能。

让我们回顾一下本节所学到的知识：

- AMQP 架构中最关键的几个组件分别是交换器、队列和绑定。
- 根据绑定规则将队列绑定到交换器上。
- 消息是发布到交换器上的。
- 有三种类型的交换器：direct、fanout 和 topic。
- 基于消息的路由键和交换器类型，服务器会决定将消息投递到哪个队列去。

2.4 多租户模式：虚拟主机和隔离

在理解了交换器、绑定和队列之后，你可能会认为自己已经掌握了 Rabbit 的所有特性。但是随着深入使用 Rabbit 之后，你会发现有一个概念我们尚未讨论过：vhost。每一个 RabbitMQ 服务器都能创建虚拟消息服务器，我们称之为虚拟主机（vhost）。每一个 vhost 本质上是一个 mini 版的 RabbitMQ 服务器，拥有自己的队列、交换器和绑定……更重要的是，它拥有自己的权限机制。这使得你能够安全地使用一个 RabbitMQ 服务器来服务众多应用程序，而不用担心你的 Sudoku（数独）应用可能会删除狗狗防丢跟踪器正在使用的队列。vhost 之于 Rabbit 就像虚拟机之于物理服务器一样：它们通过在各个实例间提供逻辑上分离，允许你为不同应用程序安全保密地运行数据。这很有用，它既能将同一 Rabbit 的众多客户区分开来，又可以避免队列和交换器的命名冲突。否则你可能不得不运行多个 Rabbit，并忍受随之而来头疼的管理问题。相反，你可以只运行一个 Rabbit，然后按需启动或关闭 vhost。

vhost 是 AMQP 概念的基础，你必须在连接时进行指定。由于 RabbitMQ 包含了开箱即用的默认 vhost："/"，因此使用起来非常简单。如果你不需要多个 vhost 的话，那么就使用默认的吧。通过使用缺省的 guest 用户名和密码 guest 就可以访问默认 vhost。为安全起见，你应该更改它（更多内容请参见第 3 章）。AMQP 的一个有趣的地方在于它并没有指定权限控制是在 vhost 级别还是在服务器端级别实现。这留给了服务器的开发者去决定。在 RabbitMQ 的例子中，权限控制是以 vhost 为单位的。

当你在 Rabbit 里创建一个用户时，用户通常会被指派给至少一个 vhost，并且只能访问被指派 vhost 内的队列、交换器和绑定。当你在设计消息通信架构时，记住 vhost 之间是绝对隔离的。你无法将 vhost banana_tree 上的交换器绑定到 vhost oak_tree 中的队列去。事实上，这既保证了安全性，又确保了可移植性。假设你为重要的银行应用设计了支票兑现层，并使用了自己的 vhost。最初，你可能把这个 vhost 同其他 vhost 一起运行在同一个 Rabbit 上。直到有一天客户们开始兑换数以百万计的支票——对你来说是好事，但是对 Rabbit 服务器来说可就惨了。支票兑换需要运行在轻负载的 Rabbit 服务器上。如果支票兑现层使用的是默认 vhost 的话，则你不得不担心命名冲突（队列和交换器）。不过既然它有自己的 vhost，你就能安全地迁移到新的 Rabbit 服务器上，然后马上开始处理新的负载而不会有任何命名冲突。因此，我们强烈推荐你仔细甄别基础架构（譬如 Web 日志）中的通用功能群组，并为它们分别分配各自的 vhost。同时，请记住，当你在 RabbitMQ 集群上创建

vhost 时，整个集群上都会创建该 vhost。vhost 不仅消除了为基础架构中的每一层运行一个 RabbitMQ 服务器的需要，同样也避免了为每一层创建不同集群。

我们讨论了 vhost 带来的所有巨大益处，那么如何创建它们呢？ vhost 和权限控制非常独特，它们是 AMQP 中唯一无法通过 AMQP 协议创建的基元（不同于队列，交换器和绑定）。对于 RabbitMQ 来说，你需要通过 RabbitMQ 的安装路径下 ./sbin/ 目录中的 `rabbitmqctl` 工具来创建。通过简单地运行 `rabbitmqctl add_vhost[vhost_name]` 就可以创建一个 vhost。其中 [vhost_name] 就是你想要创建的 vhost。删除 vhost 同样简单：`rabbitmqctl delete_vhost[vhost_name]`。一旦 vhost 创建成功之后，你就可以连接上去并开始添加队列和交换器了。如果你想要知道特定 Rabbit 服务器上运行着哪些 vhost 时，运行 `rabbitmqctl list_vhosts` 即可。之后你就会看见如下所示的内容：

```
$ ./sbin/rabbitmqctl list_vhosts
Listing vhosts ...
/
oak
sycamore
...done.
```

注意：通常情况下，你将在服务器上直接运行 `rabbitmqctl` 来管理自己的 RabbitMQ 节点。不过，你也可以通过指定 `-n rabbit@[server_name]` 选项来管理远程 RabbitMQ 节点。@ 符号将节点标识符（rabbit@[server_name]）分成两部分：左边的是 Erlang 应用程序名称，在这里永远都是 rabbit。右边是服务器主机名或者 IP 地址。你需要确保运行 Rabbit 节点的服务器和运行 `rabbitmqctl` 的工作站安装了相同的 Erlang cookie。更多关于 Erlang cookie 的信息，请参阅 3.4.1 节。

到目前为止，通过 vhost 你保障了队列和交换器的安全。现在我们来讨论当 Rabbit 崩溃或者重启时，如何确保关键消息不丢失。

2.5 我的消息去哪儿了呢？持久化和你的策略

关于在 Rabbit 里创建队列和交换器有个不可告人的秘密：默认情况下它们无法幸免于服务器重启。没错，重启 RabbitMQ 服务器后，那些队列和交换器就都消失

了（随同里面的消息）。原因在于每个队列和交换器的 durable 属性。该属性默认情况为 false，它决定了 RabbitMQ 是否需要在崩溃或者重启之后重新创建队列（或者交换器）。将它设置为 true，这样你就不需要在服务器断电后重新创建队列和交换器了。你也许会认为把队列和交换器的 durable 属性设置为 true 就足够可以让消息幸免于重启，但是你错了。队列和交换器当然必须被设置成 true，但光这样做还不够。

能从 AMQP 服务器崩溃中恢复的消息，我们称之为持久化消息。在消息发布前，通过把它的“投递模式”（delivery mode）选项设置为 2（AMQP 客户端可能会使用人性化的常量来代替数值）来把消息标记成持久化。到目前为止，消息还只是被表示为持久化的，但是它还必须被发布到持久化的交换器中并到达持久化的队列中才行。如果不是这样的话，则包含持久化消息的队列（或者交换器）会在 Rabbit 崩溃重启后不复存在，从而导致消息成为孤儿。因此，如果消息想要从 Rabbit 崩溃中恢复，那么消息必须：

- 把它的投递模式选项设置为 2（持久）
- 发送到持久化的交换器
- 到达持久化的队列

做到以上三点，你就不用和你的关键消息玩“躲猫猫”了。

RabbitMQ 确保持久性消息能从服务器重启中恢复的方式是，将它们写入磁盘上的一个持久化日志文件。当发布一条持久性消息到持久交换器上时，Rabbit 会在消息提交到日志文件后才发送响应。记住，之后这条消息如果路由到了非持久队列的话，它会自动从持久性日志中移除，并且无法从服务器重启中恢复。如果你使用持久性消息的话，则确保之前提到的持久性消息的那三点都必须做到位（我们再怎么强调也不为过）。一旦你从持久化队列中消费了一条持久性消息的话（并且确认了它），RabbitMQ 会在持久化日志中把这条消息标记为等待垃圾收集。在你消费持久性消息前，如果 RabbitMQ 重启的话，服务器会自动重建交换器和队列（以及绑定），重播持久性日志文件中的消息到合适的队列或者交换器上（取决于 Rabbit 服务器宕机的时候，消息处在路由过程的哪个环节）。

你可能认为自己应该为所有的消息都启用持久化消息通信。你可以这样做，但同时你也要为此付出代价：性能。写入磁盘要比存入内存中慢不止一点点，而且会极大地减少 RabbitMQ 服务器每秒可处理的消息总数。使用持久化机制而导致

消息吞吐量降低至少 10 倍的情况并不少见[2]。另外还有一点就是，持久性消息在 RabbitMQ 内建集群环境下工作得并不好。虽然 RabbitMQ 集群允许你和集群中的任何节点的任一队列进行通信，但是事实上那些队列均匀地分布在各个节点而没有冗余（在集群中任何一个队列都没有备份的拷贝）。如果运行 seed_bin 队列的集群节点崩溃了，那么直到节点恢复前，这个队列也就从整个集群中消失了（如果队列是可持久化的）。更重要的是，当节点宕机时，其上的队列也都不可用了，而且持久化队列也无法重建。这就会导致消息丢失。我们会在第 5 章更详细地讨论这一情况，并给出替代的集群方法来解决这个问题。

权衡取舍，什么情况下你应该使用持久性 / 持久化消息通信呢？首先，你需要分析（并测试）性能需求。你是否需要单台 Rabbit 服务器每秒处理 100 000 条消息呢？如果是这样的话，你应该寻找其他方式来保证消息投递（或者使用更快速的存储系统）。举例来说，生产者可以在单独的信道上监听应答队列。每次发送消息的时候，都包含应答队列的名称，这样消费者就可以回发应答以确认接收到了。如果消息应答未在合理时间范围内到达，生产者就重新发送消息。也就是说，要保证消息的投递这一关键本质决定了相对于其他类型的消息（例如日志消息）会有更低的吞吐量。因此如果持久性消息通信能够满足性能需求的话，那么用这种机制确保消息投递是极佳的方式。我们更多的是为关键消息使用持久化机制。我们只是对何种类型的内容使用持久性消息通信举棋不定。举个例子，我们运行两种类型的 Rabbit 集群：非持久化消息通信的传统 RabbitMQ 集群和持久化消息通信的活动 / 热备非集群 Rabbit 服务器（使用负载均衡）。这样做确保了为持久化消息通信处理负载不会减慢非持久化消息的处理。这也意味着 Rabbit 内建集群在节点宕机时不会让持久性消息消失。请记住 Rabbit 能帮助确保投递，但并不是万无一失的。硬盘崩溃、充满 bug 的消费者或者其他极端事件都能导致持久化消息丢失。最终确保消息安全到达都将取决于你的策略。持久化消息通信是一个很好的工具，可以帮助你完成这一点。

和消息持久化相关的一个概念是 AMQP 事务（transaction）。到目前为止，我们讨论的是将消息、队列和交换器设置为持久化。这一切都工作得很好，并且 RabbitMQ 也负责保证消息的安全。但是由于发布操作不返回任何信息给生产者，那你怎么知道服务器是否已经持久化了持久消息到硬盘呢？服务器可能会在把消息写入磁盘前就宕机了，消息因此而丢失，而你却不知道。这就是事务发挥作用的地方。

---

2 将RabbitMQ的消息存储置于SSD上的话，就可以极大地提升持久化消息通信的性能。

当继续处理其他任务前，你必须确保代理接收到了消息（并且已经将消息路由给所有匹配的订阅队列），你需要把这些行为包装到一个事务中。如果你有数据库背景的话，不要把 AMQP 事务和大多数数据库的事务概念搞混了。在 AMQP 中，在把信道设置成事务模式后，你通过信道发送那些想要确认的消息，之后还有多个其他 AMQP 命令。这些命令是执行还是忽略，取决于第一条消息发送是否成功。一旦你发送完所有命令，就可以提交事务了。如果事务中的首次发布成功了，那么信道会在事务中完成其他 AMQP 命令。如果发送失败的话，其他 AMQP 命令将不会执行。事务填补了生产者发布消息以及 RabbitMQ 将它们提交到磁盘上这两者之间“最后 1 英里”的差距。不过，还有更好的方法来填补差距。

虽然事务是正式 AMQP 0-9-1 规范的一部分，但是它们有阿喀琉斯之踵[3]：几乎吸干了 Rabbit 的性能。使用事务不但会降低大约 2~10 倍的消息吞吐量，而且会使生产者应用程序产生同步。而你使用消息通信就是想要避免同步。知晓了所有这一切之后，RabbitMQ 团队决定拿出更好的方案来保证消息投递：发送方确认模式[4]。和事务相仿，你需要告诉 Rabbit 将信道设置成 confirm 模式，而且你只能通过重新创建信道来关闭该设置。一旦信道进入 confirm 模式，所有在信道上发布的消息都会被指派一个唯一的 ID 号（从 1 开始）。一旦消息被投递给所有匹配的队列后，信道会发送一个发送方确认模式给生产者应用程序（包含消息的唯一 ID）。这使得生产者知晓消息已经安全到达目的队列了。如果消息和队列是可持久化的，那么确认消息只会在队列将消息写入磁盘后才会发出。发送方确认模式的最大好处是它们是异步的。一旦发布了一条消息，生产者应用程序就可以在等待确认的同时继续发送下一条。当确认消息最终收到的时候，生产者应用的回调方法就会被触发来处理该确认消息。如果 Rabbit 发生了内部错误从而导致了消息的丢失，Rabbit 会发送一条 nack（not acknowledged，未确认）消息。就像发送方确认消息那样，只不过这次说明的是消息已经丢失了。同时，由于没有消息回滚的概念（同事务相比），因此发送方确认模式更加轻量级，同时对 Rabbit 代理服务器的性能影响几乎可以忽略不计。

至此，你了解了 RabbitMQ 的各个部分，包括从消费者和生产者到持久化消息

---

3 阿喀琉斯之踵（Achilles' Heel），原指阿喀琉斯的脚跟。因为这是其唯一一个没有浸泡到神水的地方，所以是他唯一的弱点。后来，他在特洛伊战争中被人射中脚跟而致命。这里指事务本身具有致命的弱点。——译者注

4 发送方确认（publisher confirm）模式是RabbitMQ独有的针对AMQP的扩展（尽管其他服务器也可以自由地将其添加到AMQP实现中去）。该模式仅在RabbitMQ 2.3.1或更高版本中可用）。

通信。那么它们如何结合起来呢？一条真实消息的生命周期是怎样的呢？回答这些问题最好的方法就是，在代码层面来看看消息的一生。

2.6 把所有内容结合起来：一条消息的一生

我们已经讨论了 RabbitMQ 的历史，我们也讨论了 AMQP 及其细节，也安装了服务器；现在是时候让我们动手写点代码了。我们会演示消息的创建、发布，然后在服务器的另一端进行消费。我们会按照传统来开始自己的第一个 Hello World 例子，就这么办吧。

在本书中，我们用 PHP 和 Python 语言来编写代码。感谢 AMQP，代码示例可以轻而易举地移植到另一门库和语言（如果仔细比较 Python 代码和 PHP 代码，就会明白这一点）。

为了构建 Hello World 程序，让我们从你需要做哪些准备开始。

- Python 2.6 或更高版本——你需要 Python 解释器的最新版本。我们使用的是安装在 Mac OS X 10.6 平台的 2.6.1 版本。你可以使用包管理器在你的 Linux 平台上安装 Python。
- easy_install——这个小程序是 Python 包 `setuptools` 的一部分。它会帮你安装额外的包。
- Pika 0.9.6 或更高版本——Pika 是 rabbit 家族可爱的成员，同时也是 Rabbit 团队编写的官方 Python AMQP 库。

为了设置环境，首先需要安装 easy_install（根据操作系统的要求，你也许需要在 sudo 下运行这些命令）：

```
$ wget http://peak.telecommunity.com/dist/ez_setup.py
...
(25.9 KB/s) - ez_setup.py saved [10285/10285]

$ python ez_setup.py
...
Installed /Library/Python/2.6/site-packages/setuptools-0.6...
```

安装完 easy_install 之后，下一步安装 Pika：

```
$ easy_install pika
...
```

```
Installed /Library/Python/2.6/site-packages/pika-0.9.6-py2.6.egg
Processing dependencies for pika
Finished processing dependencies for pika
```

下一步创建文件夹存放你的示例代码：

```
$ mkdir chapter-2
$ cd chapter-2
```

让我们从发布者的代码开始。它需要完成以下任务：

- 连接到 RabbitMQ
- 获取信道
- 声明交换器
- 创建消息
- 发布消息
- 关闭信道
- 关闭连接

至此你完成了所有的设置工作，在文本编辑器里打开 hello_world_producer.py 文件，并键入以下代码。

清单 2.1 Hello World 生产者

```
import pika, sys

credentials = pika.PlainCredentials("guest", "guest")
conn_params = pika.ConnectionParameters("localhost",
                                        credentials = credentials)
conn_broker = pika.BlockingConnection(conn_params)

channel = conn_broker.channel()

channel.exchange_declare(exchange="hello-exchange",
                         type="direct",
                         passive=False,
                         durable=True,
                         auto_delete=False)

msg = sys.argv[1]
msg_props = pika.BasicProperties()
msg_props.content_type = "text/plain"

channel.basic_publish(body=msg,
                      exchange="hello-exchange",
                      properties=msg_props,
                      routing_key="hola")
```

建立到代理服务器的连接 ❶

❷ 获得信道

❸ 声明交换器

❹ 创建纯文本消息

❺ 发布消息

让我们看看这堆代码做了些什么。首先在❶处，用样板代码来连接 RabbitMQ（这里没有指定虚拟主机，使用的是默认的“/”）。RabbitMQ 默认运行在本机的5672端口上。使用默认的guest用户名和密码。然后获得信道❷和RabbitMQ进行通信。

下一步是声明交换器❸。消息将发送到这里。第一个参数是交换器名称 `hello-exchange`，第二个参数是交换器类型 direct。第一个 Boolean 标记告诉 RabbitMQ 你发送的是非 passive 模式的 `declare` 命令。这说明你想要声明交换器，而非仅仅获得它的信息。后两个标记说明你想要持久化的交换器，并且不会自动删除。

然后在第❹处你着手创建了一条消息。消息的内容是传递给 hello_world_consumer.py[5] 脚本的第一个参数。消息内容类型是 `text/plain`。

获得消息之后，你就将其通过 `basic_publish` 命令发送给 `hello-exchange` ❺。

你不需要每次都关闭连接。你可以通过一条信道 / 连接发送多条信息。发送完之后再关闭。

发布者已经准备就绪，现在让我们创建消费者。它需要执行以下任务：

- 连接到 RabbitMQ
- 获得信道
- 声明交换器
- 声明队列
- 把队列和交换器绑定起来
- 消费消息
- 关闭信道
- 关闭连接

看起来和之前很像，但事实上却不是。在之前的代码样例中，你已经完成了前三步和后两步。这里新的内容是如何声明队列，并绑定到交换器上，然后开始消费新消息。

创建一个名为 hello_world_consumer.py 的文件，并把以下代码加入进去。

如你所见，这些包含类库和定义常量的代码是一样的。你可以把这些定义放置到 config.py 文件中以避免重复代码。我们把这个作为练习留给你完成。

---

5 作者笔误，应该是hello_world_producer.py。——译者注

清单 2.2 Hello World 消费者

```
import pika
credentials = pika.PlainCredentials("guest", "guest")
conn_params = pika.ConnectionParameters("localhost",
                                        credentials = credentials)
conn_broker = pika.BlockingConnection(conn_params)
channel = conn_broker.channel()
channel.exchange_declare(exchange="hello-exchange",
                         type="direct",
                         passive=False,
                         durable=True,
                         auto_delete=False)
channel.queue_declare(queue="hello-queue")
channel.queue_bind(queue="hello-queue",
                   exchange="hello-exchange",
                   routing_key="hola")

def msg_consumer(channel, method, header, body):
    channel.basic_ack(delivery_tag=method.delivery_tag)
    if body == "quit":
        channel.basic_cancel(consumer_tag="hello-consumer")
        channel.stop_consuming()
    else:
        print body

    return
channel.basic_consume( msg_consumer,
                       queue="hello-queue",
                       consumer_tag="hello-consumer")
channel.start_consuming()
```

❶ 建立到代理服务器的连接

❷ 获得信道

❸ 声明交换器

❹ 声明队列

❺ 通过键“hola”将队列和交换器绑定起来

❻ 用于处理传入的消息的函数

❼ 消息确认

❽ 停止消费并退出

❾ 订阅消费者

❿ 开始消费

在❶处你建立了到服务器的连接，并在❷处创建了信道。在获得了连接之后，你在❸处再一次声明了交换器[6]。由于 declare 命令的语义是“如果没有就创建;否则继续”，所以这里什么都没发生。如果之后你发送 queue_bind 命令时，万一交换器并没有预先创建好。这样做为的是要避免这类错误的发生。

在❹处，你使用 AMQP 默认参数声明了一个名为 hello-queue 的队列。之后你把队列绑定到 hello-exchange ❺。你使用的路由键为 hola。

你几乎已经准备就绪开始消费消息了，不过你还需要一个用于处理消息的回调函数。在❻处你创建了该函数，它会确认消息❼。因此，RabbitMQ 才能删除消息并给消费者发送新的消息。在回调函数的末尾，你打印出了消息体。不久你就会知道❽处做了什么。

6 第一次是在hello_world_producer.py文件中声明的。——译者注

有了回调函数，你就可以发送❾basic_consume命令订阅队列了。第一个参数就是你刚编写的回调函数。接下来的参数分别是队列的名称和用于标识进程的消费者标记。RabbitMQ发送给消费者的每一条消息都会传入你的回调函数。

通过之前的命令，你已经准备好接收消息了。现在你必须要真的这样做了。你在❿处开始了一个阻塞的循环等待从信道进来的数据。如果RabbitMQ发送消息给你，pika会把消息传递给你的回调函数的。

我们还剩最后一点困惑❽没有讲到。start_consuming()是一个无尽的阻塞while循环。要想让它停止，你可在回调函数内插入一个条件判断。条件判断会检查消息体是否为'quit'，如果是的话，就会发送basic_cancel命令以结束消费（同时关闭信道和连接）。你必须提供消费者标记作为basic_cancel的参数。

现在让我们来测试一下代码吧。首先你必须启动RabbitMQ，然后打开新的终端窗口，移动到RabbitMQ的安装目录，输入以下命令：

```
$ sbin/rabbitmq-server
```

一旦RabbitMQ运行起来了，切换回之前的终端窗口，用以下命令启动消费者（见图2.7）：

```
$ python ./hello_world_consumer.py
```

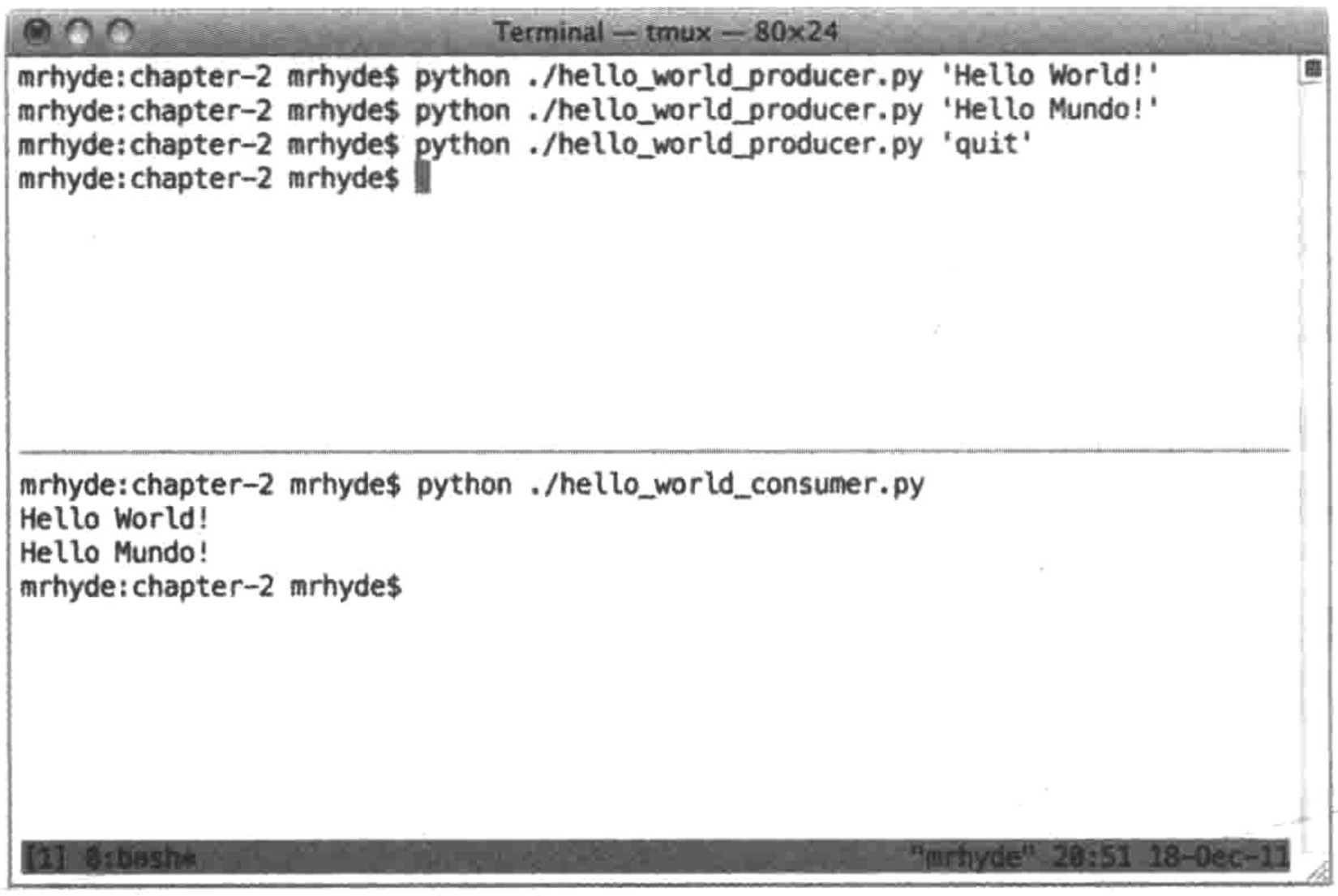

图2.7 RabbitMQ Hello World

打开一个新的终端窗口，移动到 chapter-2 文件夹，输入以下命令：

```
$ python ./hello_world_producer.py 'Hello World!'
```

如果一切正常，你应该会在运行消费者的终端窗口上看到 "Hello World!" 文本。恭喜你，一切工作正常！

现在像这样发送其他消息试试：

```
$ python ./hello_world_producer.py 'Hello Mundo!'
```

你应该可以看到："Hello Mundo!"

最后我们停止消费者：

```
$ python ./hello_world_producer.py 'quit'
```

让我们看看你刚刚做了些什么。你想要发送消息并消费它们，因此你声明了交换器以保证有地方发布消息。同时你也创建了队列并绑定到 hello-exchange 上。然后基于你在命令行输入的文本，你创建了消息实例并通过 RabbitMQ 发送出去。基于 direct 交换器类型，RabbitMQ 将消息路由到 hello-queue 上。由于在另一端有消费者在等待消费，因此 RabbitMQ 投递了消息并由回调函数进行处理。你看到此处并没有提供路由键。那是因为 AMQP 既可以很简单也可以很复杂，这取决于我们怎么使用它。对我们的例子来说空白路由键就够了。

2.7 使用发送方确认模式来确认投递

你明白了如何编写基本的消费者和生产者。但你也许还在思考如何将发送方确认模式和事务结合起来。让我们看看你如何改进 Hello World 生产者，利用 publisher confirm 来追踪消息投递[7]。在我们动手改进 Hello World 生产者使用发送方确认模式之前先看看图 2.8，它形象地展示了消息 ID 是如何指派的。

7 由于事务会影响性能，因此我们专注于使用publisher confirm作为消息投递保障的首选方案。我们将Hello World生产者的事务版本包含在了本书的在线示例中：https://github.com/rabbitinaction/sourcecode。

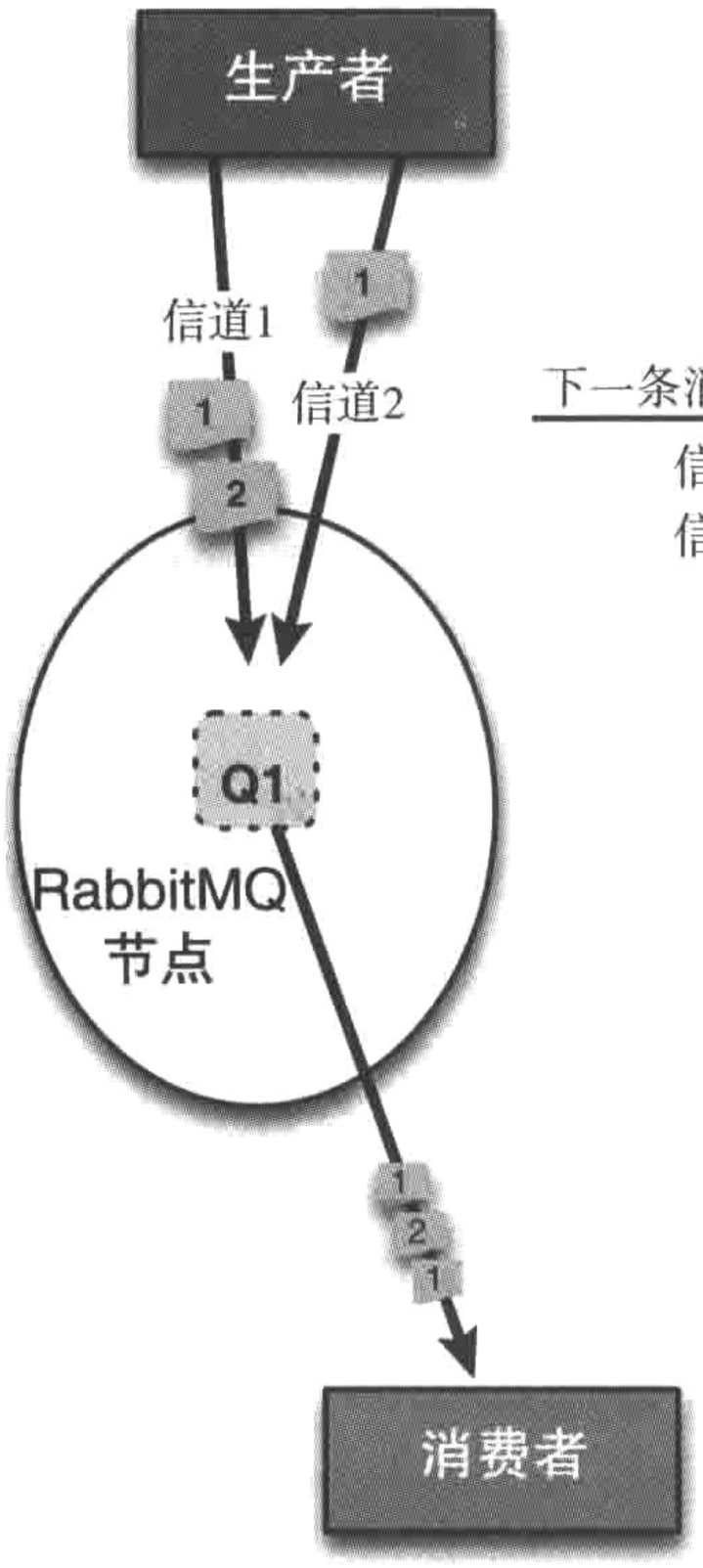

图 2.8 发布者确认消息 ID

之前我们说过，当信道设置成 confirm 模式时，发布的每一条消息都会获得唯一的 ID。这可能会让你猜测 `basic_publish` 会返回消息 ID，不过事实上消息 ID 不是这样工作的。由于一条信道只能被单个线程使用，因而可以确保信道上发布的消息都是连续的。因此，RabbitMQ 做了个简单的假设：任一信道上发布的第一条消息将获得 ID `1`，并且信道上接下来的每一条消息的 ID 都将步进 `1`。也就是说，信道上发布的第二条消息将会拥有 ID `2`,第三条消息将会拥有 ID `3`,等等。对信道来说，消息 ID 是唯一的。所以一旦信道关闭后，你将无法追踪发布在该信道上任何未完成的发送方确认消息状态。这意味着 RabbitMQ 不必告诉你刚发布的消息的 ID；你在应用程序内部通过一个计数器自己来跟踪。每次应用程序信道发布消息时，你需要把计数器加 `1`。同时，由于每条信道的消息 ID 都是从 `1` 计数的，因此如果你同时拥有众多运行的信道的话，就需要为每条信道分别维护一个内部消息 ID 计数器。

清单 2.3 带有确认功能的 Hello World 生产者

```
import pika, sys
from pika import spec
credentials = pika.PlainCredentials("guest", "guest")
conn_params = pika.ConnectionParameters("localhost",
                                        credentials = credentials)
conn_broker = pika.BlockingConnection(conn_params)
channel = conn_broker.channel()
def confirm_handler(frame):
    if type(frame.method) == spec.Confirm.SelectOk:
       print "Channel in 'confirm' mode."
    elif type(frame.method) == spec.Basic.Nack:
       if frame.method.delivery_tag in msg_ids:
           print "Message lost!"
    elif type(frame.method) == spec.Basic.Ack:
       if frame.method.delivery_tag in msg_ids:
           print "Confirm received!"
           msg_ids.remove(frame.method.delivery_tag)

channel.confirm_delivery(callback=confirm_handler)

msg = sys.argv[1]
msg_props = pika.BasicProperties()
msg_props.content_type = "text/plain"
msg_ids = []
channel.basic_publish(body=msg,
                      exchange="hello-exchange",
                      properties=msg_props,
                      routing_key="hola")
msg_ids.append(len(msg_ids) + 1)
channel.close()
```

❶ 发送方确认模式处理器

❷ 将信道设置为confirm模式

❸ 重设消息ID追踪器

❹ 发布信息

❺ 将ID添加到追踪列表中

这段代码和最初的 Hello World 生产者很像，不过现在你增加了一个回调函数❶ `confirm_handler`。该函数会在应用程序接收到发送方确认消息时调用。之后你告诉 Pika 将信道设置成 confirm 模式❷，并用 `confirm_handler` 作为回调函数接收到达的发送方确认消息。一旦信道设置成处理发送方确认消息之后，你需要设置内部列表（`msg_ids`）来追踪信道上的消息 ID ❸，并发布消息❹。最后，消息发布一完成，你就要将代表消息 ID 的 `msg_id_no` 增加 1 ❺。有关 confirm 的有趣魔法都包含在 `confirm_handler` 里了。

当 `confirm_handler` 接收到一条发送方确认消息时，它首先检查确认类型是否是 `Confirm.SelectOk`：

```
if type(frame.method) == spec.Confirm.SelectOk:
    print "Channel in 'confirm' mode."
```

当第一次将信道设置为 confirm 模式时，RabbitMQ 会发送一条 Confirm.SelectOk 类型的确认消息。这并不是对消息的确认，而是通知你信道已经准备就绪来接收发送方确认消息了。如果确认消息不是 Confirm.SelectOk，那么 confirm_handler 检查是否为 Basic.Nack 确认消息：

```
elif type(frame.method) == spec.Basic.Nack:
    if frame.method.delivery_tag in msg_ids:
    print "Message lost!"
```

你也许记得我们解释发送方确认模式时说过，Basic.Nack 指明了消息由于 RabbitMQ 内部错误而丢失了。但在你将消息处理为丢失之前，先检查一下 frame.method.delivery_tag 以确保消息确实是丢失了。如果消息 ID 和发布的 ID 一致，那么你得告诉用户这条消息丢失了。在更复杂的应用中，这里还是放置重发丢失消息代码的地方。最终，如果确认消息既不是 Confirm.SelectOk，也不是 Basic.Nack，那么需要检查它是否是 Basic.Ack。

```
elif type(frame.method) == spec.Basic.Ack:
```

假设该消息为发送方确认模式的确认消息（Basic.Ack），则你需要确保消息 ID 在已发布的消息 ID 列表中：

```
if frame.method.delivery_tag in msg_ids:
    print "Confirm received!"
    msg_ids.remove(frame.method.delivery_tag)
```

如果这个确认消息的消息 ID 是所追踪的 msg_ids 中的一个，那么你向用户确认该消息已经成功到入队，然后将这条消息的 ID 从等待投递确认的 ID 列表中删除。这比最初版本的 Hello World 生产者稍微复杂一点，不过也只用了 12 行代码就为消息发布增加了追踪投递的能力。更令人印象深刻的是，这段简单的代码可以追踪每分钟数以百万计的消息投递。这显示了发送方确认模式比 AMQP 事务的表现要好得多。

2.8 总结

本章我们讲解了很多内容。你不仅拥有了消息通信基础来构建任何你能想象得到的应用程序，同时也拥有了真实存在的生产者和消费者，特别是生产者还能追踪消息投递！在接下来的几章，我们会讲述构建消息通信应用程序的具体模式，它能

让你的应用程序优雅、高效和强大。但就目前你已经学到的知识，你可以立即为那个令你生不如死的分布式 Twitter 克隆开始编码了。在深入编码细节之前，让我们先来看看如何更专业地管理 RabbitMQ 服务器。举例来说，让任何人都能连接到你的虚拟主机上，然后发布消息给任一交换器，这可不是什么好主意。这就是下一步我们将要向你展示的，如何通过设置权限控制去避免这种类型的问题。因此，让我们看看如何启动、停止，以及管理 RabbitMQ 吧！

3 运行和管理Rabbit

本章要点

- 服务器管理——启动和停止节点
- 权限配置
- 使用统计
- RabbitMQ和Erlang问题疑难解答

我们花了大把的时间在 AMQP 消息通信概念以及如何运行基本安装的 RabbitMQ 上。现在我们准备对 RabbitMQ 日常基础管理一探究竟。了解如何让 RabbitMQ 在工作站上运行起来是一回事儿，但你知道如何干净利落地停止它吗？你知道如何限制 RabbitMQ 的内存消耗以致不会让同一台服务器上的其他应用程序“饿死”吗？当 RabbitMQ 从开发环境转入生产环境时，你就会面对这些问题。在本章中我们会向你展示如何运行管理一流的 RabbitMQ 产品环境，以免你陷入困境。

其中我们会涉及不同的东西：

- Erlang 如何运作的一些背景，包括那些神秘的 Erlang cookie
- 通过 RabbitMQ 的权限系统来控制用户访问

- 使用命令行工具查看虚拟机、队列、交换器和绑定的状态
- 当遇到可怕的 Erlang 错误消息，例如 "badrpc,node down" 时，该怎么办
- 如何解读各种 RabbitMQ 日志文件

在我们结束的那一刻，你将成为一流的 RabbitMQ 管理员，并准备去解决 Rabbit 生产环境问题。让我们从基础知识开始，一步步深入学习如何管理 RabbitMQ 服务器。

3.1 服务器管理

你以前在其他产品上的使用经验可能无法作用于如何高效地运行 RabbitMQ 服务器之上。主要原因在于 RabbitMQ 是用 Erlang 编写的，而 Erlang 有自己的一套方式。大多数情况下，这是件好事。Erlang 天生就能让应用程序无须知道对方是否在同一台机器上即可相互通信。对 RabbitMQ 来说，这让集群和可靠的消息路由变得简单。但是为了达到“简单分布式”，需要两个你可能不太熟悉的概念：Erlang 节点和 Erlang 应用程序。不用担心；这些概念并没有听起来这么难以理解。如果你熟悉 Java 虚拟机（JVM）的话，两者是很相似的。本节会教你如何启动和停止 RabbitMQ 节点以及如何使用 RabbitMQ 配置文件。事不宜迟，我们先来看看什么是节点以及如何启动它。

3.1.1 启动节点

到目前为止，我们经常使用术语节点来指代 RabbitMQ 服务器实例。事实上，节点描述的是一个 Erlang 节点运行着一个 Erlang 应用程序。提到 Erlang 的时候别紧张——你不用成为一个 Erlang 爱好者并明白它的工作原理。这和 JVM 的原理是非常相似的。

当你运行 Java 程序时，JVM 的一个实例就启动了，并且开始执行指定的 Java 程序。与之相似，Erlang 也有虚拟机，而虚拟机的每个实例我们称之为节点（node）。这里的节点非常特殊。不同于 JVM，多个 Erlang 应用程序可以运行在同一个节点之上。更重要的是，节点之间可以进行本地通信（不管它们是否真的在同一台服务器上）。举例来说，由于 Erlang 的魔力，一个运行在 `asparagus` 节点上的应用程序可以调用 `artichoke` 节点上的应用程序方法，就好像这些函数是本地方法一样。同

时，如果应用程序由于某些原因（例如 RabbitMQ 崩溃了）崩溃了，Erlang 节点会自动尝试重启应用程序（前提是 Erlang 本身没有崩溃）。当我们开始讨论插件和集群的时候，事情就会变得有趣起来。现在重要的是记住当我们谈到 RabbitMQ 节点时，我们指的是 RabbitMQ 应用程序和其所在的 Erlang 节点（见图 3.1）。

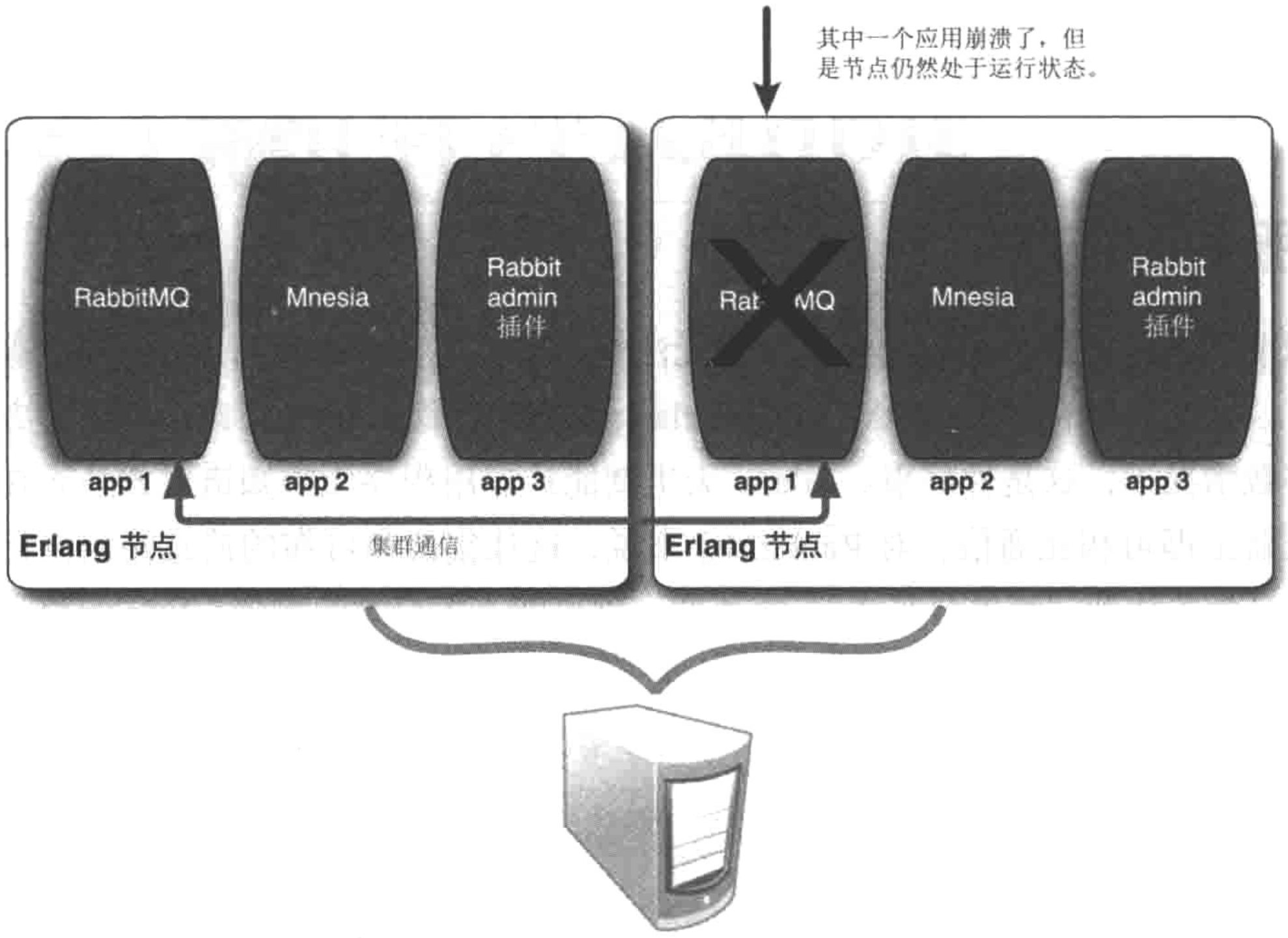

图 3.1 理解 Erlang 节点和应用

值得庆幸的是，RabbitMQ 使得启动 Erlang 节点和 Rabbit 应用程序很简单，只需一个步骤即可。在你的 RabbitMQ 安装目录下找到 `./sbin` 目录，运行 `./rabbitmq-server`[1]。当查看控制台时，你会发现不同的 RabbitMQ 子系统运行起来了，并准备好处理消息。如果在启动过程中遇到了任何错误，那么就检查一下 RabbitMQ 日志。通常情况下，可以在 `/var /log/rabbitmq/` 目录下找到名为 `rabbit@[hostname].log` 的日志文件（文件名中的“rabbit”部分是运行 RabbitMQ 的默认 Erlang 节点名称）。你也可以通过增加 `-detached` 参数的方式启动 Rabbit 节点，以守护程序的方式在后台运行：`./rabbitmq -server -detached`。这就是启动单

1 如果你是通过特定于操作系统的包来安装RabbitMQ的话（譬如，RPM），我们建议你使用包安装的start/stop init.d脚本。

一 RabbitMQ 节点的所有内容了。现在节点已经启动了，我们怎么停止它呢?

3.1.2 停止节点

当谈及停止 RabbitMQ 时，有两种方式：干净的方式和肮脏的方式。当运行 RabbitMQ 连接到控制台时，在你按下 CTRL-C 组合键时看到以下内容，也许会感到迷惑：

```
BREAK: (a)bort (c)ontinue (p)roc info (i)nfo (l)oaded
       (v)ersion (k)ill (D)b-tables (d)istribution
```

我的天哪，这都是些什么？你想要的只是停止 RabbitMQ 而已。你在控制台上看到的是 Erlang 节点正询问你是想要关闭应用程序还是整个节点；或者这只是一个失误，因为你想要继续运行。通常来讲，你想要关闭整个节点，所以 `abort` 就是你要找的答案。但是有更好的方法来停止 RabbitMQ——这个方法会通知 RabbitMQ 干净地关闭，并保护好那些持久化队列。

`rabbitmqctl` 是几乎所有 RabbitMQ 管理需要的一站式解决方案。你已经见过了如何使用它来创建 vhost 和列出所有 vhost。它也能帮助你停止 RabbitMQ。当在 RabbitMQ 安装目录下运行 `./sbin/rabbitmqctl stop` 时，`rabbitmqctl` 会和本地节点通信并指示其干净地关闭。你也可以指定关闭不同的节点，包括远程节点。只需传入 `-n rabbit@[hostname]` 选项即可。在 RabbitMQ 日志中会看到以下内容：

```
=INFO REPORT====
    application: rabbit
    exited: stopped
    type: permanent

=INFO REPORT====
    application: mnesia
    exited: stopped
    type: permanent

=INFO REPORT====
    application: os_mon
    exited: stopped
    type: permanent
```

当看到 `rabbit`、`mnesia` 和 `os_mon` 停止了，Rabbit 节点也就完全关闭了。如果你是通过包管理系统例如 Ubuntu 上的 APT 进行 RabbitMQ 安装的话，你同样可以通过安装在 `/etc/init.d/` 下的 RabbitMQ startup/shutdown 脚本来进行操作。可

以通过这个脚本来完成关闭任务。这时，整个 RabbitMQ 节点包括 Erlang 都关闭了。有时候，你只想停止 RabbitMQ 应用程序，而让 Erlang 实例继续运行。让我们看看如何做到这一点。

3.1.3 关闭和重启应用程序：有何差别

到目前为止，我们讨论了如何停止整个 RabbitMQ 节点（应用程序和 Erlang 节点一起）。但有些时候你只想要重启 RabbitMQ 应用程序，而同时保持 Erlang 节点运行。这有什么好处呢？对集群来说，这种做法是必需的。`rabbitmq-server` 同时启动了节点和应用程序，因为它把 RabbitMQ 应用程序预先配置成了独立运行模式。为了把节点加入现有的集群当中，你需要做的是停止应用程序，把节点重置为原始状态。这样节点就准备好加入集群了。如果你使用 `./rabbitmqctl stop` 命令的话，就会把应用程序和节点同时关闭。这时如果运行 `./rabbitmq-server`，由于独立运行模式的关系，又会迫使你把应用程序和节点同时启动起来。另外还有一点就是，你会在同一节点上运行除了 RabbitMQ 之外的其他 Erlang 应用程序，这使得停止整个节点是不可取的。

停止 RabbitMQ 轻而易举。只需运行 `./rabbitmqctl stop_app` 即可。Rabbit 日志会显示和之前关闭整个节点时一样的关闭消息。

我们已经学习了怎样启动和停止 RabbitMQ。但是，如何配置 RabbitMQ 让它不会吞噬整台服务器的 RAM？怎样更改 RabbitMQ 监听的端口呢？这就是 RabbitMQ 配置文件起作用的地方了。

3.1.4 Rabbit 配置文件

像大多数服务器应用程序一样，RabbitMQ 允许你设置系统范围的可调参数并通过配置文件进行设置。典型地，该配置文件位于 `/etc/rabbitmq/rabbitmq.config`。不过，文件位置可以通过 `rabbitmq-server` 脚本对 `CONFIG_FILE` 环境变量进行设置。你会发现 `rabbitmq.config` 的文件格式有点吓人：

```
[ {mnesia, [{dump_log_write_threshold, 1000}]},
{rabbit, [{vm_memory_high_watermark, 0.4}]} ].
```

你看到的本质上是原始的 Erlang 数据结构。但是如果你对 Python、JavaScript 或者其他任何现代编程语言熟悉的话，一旦对它进行分解之后，就很容易理解了。

让我们把配置格式化一下，让它变得更为友好：

```
1) [
2)     {mnesia, [{dump_log_write_threshold, 1000}]},
3)     {rabbit, [{vm_memory_high_watermark, 0.4}]}
4) ].
```

这样看来就好多了。你会发现 RabbitMQ 配置文件事实上是一个包含了嵌套哈希表（字典或者命名数组）的数组。第一行和第四行分别开启和关闭了配置数组。通过外部配置数组，每个 Erlang 应用程序会有自己的哈希表来配置选项（此处我们有两个应用）。`mnesia` 指的是 Mnesia 数据库配置选项（Mnesia 是 RabbitMQ 用来存储交换器和队列元数据的）。`rabbit` 指的是 RabbitMQ 特定的配置选项。每个选项都表达为这种形式：{[option_name], [option_value]}。例如，`{dump_log_write_threshold, 1000}` 更改了 Mnesia 把条目从仅限追加的日志文件刷出到真实数据库文件的频度。为了添加另一个 Mnesia 配置选项，只需增加一个 {[option_name], [option_value]} 条目，该条目需要和最后一个之间以逗号隔开。

注意：RabbitMQ 中的每个队列、交换器和绑定的元数据（除了消息的内容）都是保存到 Mnesia 的。Mnesia 是内建在 Erlang 的非 SQL 型数据库。Mnesia 通过将 RabbitMQ 元数据首先写入一个仅限追加的日志文件，以确保其完整性。然后，它再定期将日志内容转储到真实的 Mnesia 数据库文件中。如果你对 MySQL 的 InnoDB 这样的日志型数据库，或者像 XFS 这样的日志文件系统的工作方式熟悉的话，它们在概念上是相同的。Mnesia 的 `dump_log_write_threshold` 选项控制着转储的频度。将其设置成 1000 就告诉 Mnesia，每 1000 个条目就转储日志内容到数据库文件。

Rabbit 配置文件格式了解清楚了，那么你可以更改哪些真实的选项呢？如表 3.1 和表 3.2 所示，有些是关于 Mnesia 数据库的，有些是直接设置 Rabbit 的。

表 3.1 Mnesia 配置选项

选项名称 值类型	默认值	描述
`dump_log_write_threshold` 整型	100	将仅限追加的日志内容刷出 / 转储至真实数据库文件的频度。它明确指定了在转储操作发生前，必须有多少个条目存储在日志中。设置更高的数值将减少 I/O 负载并增加持久化消息的性能

表 3.2 Rabbit 配置选项

选项名称 值类型	默认值	描述
tcp_listeners {"ip 地址 ", 端口号 } 数组	[{"0.0.0.0", 5672},]	定义了 RabbitMQ 应该监听的非 SSL 加密通信的 IP 地址和端口
ssl_listeners {"ip 地址 ", 端口号 } 数组	空	定义了 RabbitMQ 应该监听的 SSL 加密通信的 IP 地址和端口
ssl_options {" 键 ", 值 } 数组	空	指定 SSL 相关的选项。有效的选项有 cacertfile（CA 证书文件）、certfile（服务器证书文件）、keyfile（服务器密钥文件）和 fail_if_no_peer_cert（需要客户端安装有效证书：True/False）
vm_memory_high_watermark 十进制百分数	0.4	控制 RabbitMQ 允许消耗的内存。它以十进制数值的形式明确了 Rabbit 允许使用的安装内存百分比（0.4=40%）
msg_store_file_size_limit 整型（字节）	16777216	RabbitMQ 垃圾收集存储内容之前，消息存储数据库的最大大小
queue_index_max_journal_entries 整型	262144	在转储到消息存储数据库并提交之前，消息存储日志里的最大条目数

虽然配置文件允许更改 RabbitMQ 运作的方方面面，但有一件事配置文件做不了：对 RabbitMQ 的访问控制。RabbitMQ 拥有整个专业子系统专门负责权限。让我们从学习如何创建用户开始来一步步熟悉权限吧。

3.2 请求许可

如果你熟悉不同操作系统的访问控制列表的话，那么理解 RabbitMQ 的权限系统对你来说易如反掌。就像大多数权限系统那样，首先创建用户，然后为其赋予权限，如图 3.2 所示。

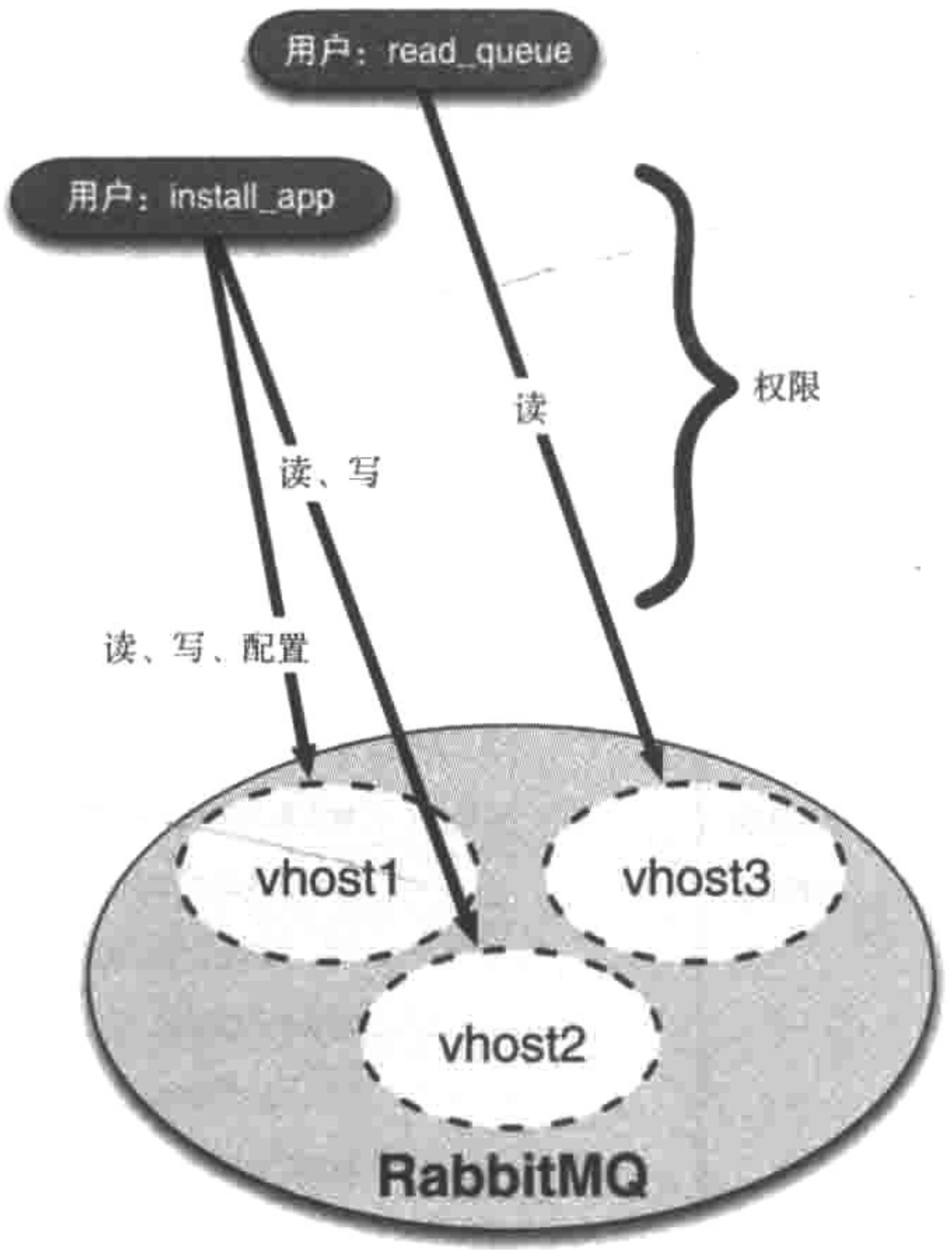

图 3.2 RabbitMQ 权限工作原理：用户可以为连接到 RabbitMQ 主机的应用程序设置不同级别的权限（读、写、和 / 或配置）

RabbitMQ 权限系统一个好的地方在于单个用户可以跨越多个 vhost 进行授权。当应用程序需要跨越多个安全域进行通信时（使用虚拟机进行隔离），这会极大地方便访问控制的管理。废话少说，让我们创建用户吧！

3.2.1 管理用户

在 RabbitMQ 中，用户是访问控制的基本单元。针对一到多个 vhost，其可以被赋予不同级别的访问权限，并使用标准的用户名 / 密码对来认证用户。对用户的增加、删除以及列出列表，都非常简单。这些操作都是通过 `rabbitmqctl` 完成的。让我们为 check-cashing 应用程序创建一个新用户吧。

从 RabbitMQ 安装目录下的 `./sbin` 运行如下命令：

```
$ ./rabbitmqctl add_user cashing-tier cashMe1
Creating user "cashing-tier" ...
...done.
```

这样就创建了一个新的 Rabbit 用户，用户名为 `cashing-tier`，密码是 cashMe1。如果想要删除该用户，则只需简单运行如下命令：

```
$ ./rabbitmqctl delete_user cashing-tier
Deleting user "cashing-tier" ...
...done.
```

请注意，当你删除用户的时候，任何引用该用户的访问控制条目都会从 Rabbit 权限数据库中自动删除。同时，rabbitmqctl 也不会警告你与用户相关的访问控制条目也会一并被删除。所以请谨慎删除用户；否则你会发现不得不重建一大堆访问控制条目。

你经常会需要知道在当前 Rabbit 服务器上存在哪些用户。通过传入 list_user 命令到 rabbitmqctl 就可以进行查看：

```
$ ./rabbitmqctl list_users
Listing users ...
cashing-tier
guest
...done.
```

你也许会问自己："不错。不过，我如何更改已经存在的用户的密码呢？我需要重新创建用户？那不是会丢失用户的访问控制条目吗？"别害怕，rabbitmqctl 早就考虑到了。只要简单运行 change_password 命令，指明需要更改密码的用户名称和新密码即可：

```
$ ./rabbitmqctl change_password cashing-tier compl3xPassword
Changing password for user "cashing-tier" ...
...done.
```

你看！ cashing-tier 的新密码已经变成 compl3xPassword 了。你会发现，RabbitMQ 管理用户是很简单的。当你开始为用户赋予访问控制条目时，复杂性才真正体现出来。

3.2.2 Rabbit 的权限系统

从 1.6.0 版本开始，RabbitMQ 实现了一套访问控制列表（ACL）风格的权限系统。在这以前，用户只能针对整个 vhost 进行授权或拒绝访问（对于已授权的 vhost，用户可以做任何事情）。新的权限系统允许大量细粒度控制，同时可以授予用户读、写和配置权限。那么这三者之间有何差异？

- 读——有关消费消息的任何操作，包括"清除"整个队列（同样需要绑定操作的成功）

- 写——发布消息（同样需要绑定操作的成功）
- 配置——队列和交换器的创建和删除

表 3.3 展示了不同 AMQP 命令的列表和对应的权限（当前 RabbitMQ 版本为 2.0.0）。

表 3.3 AMQP 操作到 RabbitMQ 权限的映射关系

AMQP 命令	配置	写	读
exchange.declare	exchange		
exchange.delete	exchange		
queue.declare	queue		
queue.delete	queue		
queue.bind		queue	exchange
basic.publish		exchange	
basic.get			queue
basic.consume			queue
queue.purge			queue

每一条访问控制条目由以下四部分组成：

- 被授予访问权限的用户。
- 权限控制应用的 vhost。
- 需要授予的读 / 写 / 配置权限的组合。
- 权限范围——权限控制仅应用于客户端命名的队列 / 交换器呢，还是仅用于服务器端命名的队列 / 交换器？抑或两者兼顾？客户端命名意味着由你的应用程序设置了交换器 / 队列的名称；服务器端命名意味着你的应用程序不提供名字而是让服务器随机指派。

谨记：访问控制条目是无法跨越 vhost 的。举例来说，如果你想要给用户 cashing-tier 在 vhost oak 和 vhost sycamore 上赋予相同的权限，那么你必须创建两份访问控制条目（每个 vhost 一份）。理论讲解到此为止。让我们动手创建一个访问控制条目吧！

在这个例子里，假设你有名为 sycamore 的 vhost，你想要授予 cashing-tier 完全的访问权限（配置、写和读权限）。你需要 rabbitmqctl 的 set_permissions 命令来完成：

```
$ ./rabbitmqctl set_permissions -p sycamore \
cashing-tier ".*" ".*" ".*"
```

```
Setting permissions for user "cashing-tier" in vhost "sycamore" ...
...done.
```

让我们把 set_permissions 命令分解开来，一块块分开看：

- -p sycamore——这告诉了 set_permissions 条目应该应用到哪个 vhost 上。
- cashing-tier——被授予权限的用户。
- ".*" ".*" ".*"——这是授予的权限。这些值分别映射到配置，以及写和读。

权限值是这个命令中最有趣的部分了。三个值中的每一个都是正则表达式。在这里，使用 ".*" 指代所有权限（配置，以及写和读）。".*" 意味着匹配任何队列或者交换器名字。这样就允许 cashing-tier 对任何队列或者交换器执行配置、写和读命令了。另一个例子可能会解释得更清楚。

假设你为 cashing-tier 授予在 oak vhost 上的权限。你想要允许该用户对任何队列或者交换器可执行读操作，同时限制其只能对名字以 checks- 开始的队列和交换器允许写操作。同时，你想要完全阻止配置操作。为达目的，你需要设计三个正则表达式[2]：

- ".*" 匹配任何队列和交换器
- "checks-.*" 只匹配名字以“checks-”开头的队列和交换器
- "" 不匹配队列和交换器（这就是如何对用户拒绝指定的权限）

把所有整合起来，像这样执行 set_permissions：

```
$ ./rabbitmqctl set_permissions -p oak \
-s all cashing-tier "" "checks-.*" ".*"
Setting permissions for user "cashing-tier" in vhost "oak" ...
...done.
```

你可以通过使用 rabbitmqctl 的 list_permissions 命令来验证权限是否正确地赋予给了 oak vhost。

```
$ ./rabbitmqctl list_permissions -p oak
Listing permissions in vhost "oak" ...
cashing-tier             checks-.*       .*          all
...done.
```

2 RabbitMQ权限支持标准的Perl兼容的正则表达式（PCRE）语法。

从显示的结果中可以看到，在 cashing-tier 和 "checks-.*" 之间的空白列就是你为设置配置权限传入的空白（""）值。你会发现，权限配置正确地应用到了 oak vhost 上，同时队列 / 交换器也如你所愿被保护起来了。如果想要删除这些权限控制，该怎么办呢？你可以通过使用 rabbitmqctl 的 clear_permissions 命令移除一个用户在任何 vhost 上的权限：

```
$ ./rabbitmqctl clear_permissions -p oak cashing-tier
Clearing permissions for user "cashing-tier" in vhost "oak" ...
...done.
```

通过运行 list_permissions 命令，你会看到权限都已经被移除干净了。

```
$ ./rabbitmqctl list_permissions -p oak
Listing permissions in vhost "oak" ...
...done.
```

值得注意的是，clear_permissions 命令会移除用户在指定 vhost 上的所有权限。如果你只想要修改用户现存的权限，则只需设置新的权限值并执行 set_permissions 即可。如果你想要查看用户在 RabbitMQ 服务器所有 vhost 上的权限，那么请使用 rabbitmqctl 的 list_user_permissions 命令：

```
$ ./rabbitmqctl list_user_permissions cashing-tier
Listing permissions for user "cashing-tier" ...
oak             checks-.*           .*          all
sycamore        .*      .*          .*          all
...done.
```

RabbitMQ 的权限容易创建且非常灵活。这种灵活性允许你为 vhost 创建复杂的权限结构。当你需要的时候，你会从中收益；但是如果你把它们设置得非常复杂的话，它们就很难理解。如果可能的话，尽量使用 vhost 分隔作为你的主要手段来确保应用程序的安全，并最小化每个 vhost 的访问控制条目。这样会帮助你避免难以调试的意外权限行为。

现在你能够连接并确保 Rabbit 服务器的安全，你也许想看看服务器内部到底发生了什么。也许你需要知道有多少条资源调配消息在用户创建队列中。也许你想知道交换器是否存在于正确的 vhost 上。一个健康的 RabbitMQ 服务器的最重要的组成部分之一就是能够监控其内部状态。因此，让我们看看如何检查自己的 Rabbit 服务器！

3.3 检查

到目前为止，你已经学习了 RabbitMQ 管理——如何启动和停止服务器，添加用户，管理权限，等等——但是如果你想要检查服务器状态，或者有多少个队列时，你该如何做呢？在之前章节里创建的 logs-exchange 是 topic 类型还是 fanout 类型？ msg-inbox-logs 队列上是否还有等待消费的消息呢？所有这些问题都能由强大的 rabbitmqctl 来回答！

3.3.1 查看数据统计

你可能已经注意到当我们之前运行 rabbitmqctl 脚本时，它接收了许多选项和命令。其中一个你经常见到的是 -p 选项。它指明了虚拟主机或者路径信息。如果忽略该选项，rabbitmqctl 会将“/”作为默认路径。

为了通过 rabbitmqctl 进行实验操作，我们需要先准备测试环境。让我们先创建一个交换器并将其绑定到一对队列上，这样你就有一些样板数据进行测试了。你应该知道本书的大部分例子都是用 PHP 或 Python 编写的。你已经设置好了 Python，所以现在该用 PHP 来编写这段简单的脚本了。让我们看看开发环境还需要安装些什么。首先你需要适合你的操作系统的 PHP 包。由于 PHP 非常流行，因此应该不会有什么问题。有可能你的操作系统已经预先安装了 PHP。如果你仍然需要自行安装，则我们建议你选择 5.3.x 中的一个版本。因为 PHP 自从 5.3 版后在内存管理和垃圾收集上有巨大的改进。在 PHP 安装好后，下载 PHP AMQP 库，这在整本书的讲解中都会用到。这个库叫作 php-amqplib，可以从 https://github.com/videlalvaro/php-amqplib 获得。为你的 PHP 例子创建文件夹，然后将库也下载到此处：

```
$ mkdir php
$ cd php
$ wget https://github.com/videlalvaro/php-amqplib/tarball/v1.0 \
 --no-check-certificate
$ tar -xzvf v1.0
$ mv videlalvaro-php-amqplib-b0b8696 php-amqplib
```

首先创建一个名为 php 的文件夹，然后 cd 到该文件夹下。之后通过 wget 命令从 GitHub 获取库压缩文件并解压缩。最后你把库移动到最合适的位置，这样之后可以少打点字。

现在 php-amqplib 已经下载好了，创建 PHP 脚本来初始化队列和交换器吧。

新创建一个名为 rabbitmqctl-examples.php 的文件，并把以下清单中的代码添加进来。

清单 3.1 rabbitmqctl 示例

```
<?php
require_once('./php-amqplib/amqp.inc');
define('HOST', 'localhost');
define('PORT', 5672);
define('USER', 'guest');
define('PASS', 'guest');

$conn = new AMQPConnection(HOST, PORT, USER, PASS);
$channel = $conn->channel();

$channel->exchange_declare('logs-exchange',
 'topic', false, true, false);

$channel->queue_declare('msg-inbox-errors',
  false, true, false, false);

$channel->queue_declare('msg-inbox-logs',
  false, true, false, false);

$channel->queue_declare('all-logs', false,
  true, false, false);

$channel->queue_bind('msg-inbox-errors',
  'logs-exchange', 'error.msg-inbox');

$channel->queue_bind('msg-inbox-logs',
  'logs-exchange', '*.msg-inbox');

?>
```

❶ 获得连接和信道

❷ 声明交换器

❸ 声明队列

❹ 把队列绑定到交换器上

这段脚本非常简单。我们不会深入探讨这里调用的 AMQP 方法，因为我们已经花了整整一个章节专门讲 AMQP 的使用。现在让我们尝试理解这段脚本的基础知识。首先你连接到代理服务器并获得了一条信道❶，这样你就可以和 RabbitMQ 进行通信了。获得信道之后，你声明了一个叫作 logs-exchange ❷的交换器，然后创建了三个队列：msg-inbox-errors、msg-inbox-logs 和 all-logs ❸。最后，你将这三个队列中的两个绑定到了 logs-exchange 上，第一处使用的是 error.msg-inbox 绑定规则，第二处使用的是 *.msg-inbox 绑定规则❹。这段脚本的目的是当你运行 rabbitmqctl 命令列出交换器、队列和绑定时，会显示相应的内容。确保 require_once 这行代码所指向的位置你安装了 amqp.inc 库。使用以下命令执行代码：

```
$ php ./rabbitmqctl-examples.php
```

如果一切正常，那么你就准备好用 rabbitmqctl 进行实验吧。

列出队列和消息数目

就让我们从 list_queues 命令开始吧。在终端上，更改当前目录到 sbin 文件夹下（或者到 rabbitmqctl 所在的目录）并输入：

```
$ ./rabbitmqctl list_queues
```

该命令的输出依赖于服务器上已声明的队列，但是你看到的应该和这个很像：

```
Listing queues ...
msg-inbox-logs 0
msg-inbox-errors 0
all-logs 3
...done.
```

在这里你可以看到队列名字以及在每个队列上的消息总数。如我们所说的，这些都是默认 vhost 上的信息。如果你想要获得另一个 vhost 信息的话，则可尝试像这样添加 -p 选项：

```
$ ./rabbitmqctl list_queues -p sycamore
```

这样就会展示在 sycamore vhost 上的队列了。

现在让我们看看这个命令的其他选项。如果你运行 rabbitmqctl 命令而不携带任何选项的话，就会展示帮助信息。在帮助文本中，你会看到这个：

```
list_queues      [-p <VHostPath>] [<QueueInfoItem> ...]
```

你已经知道 VHostPath 的含义。往下滚动一点你就会看到 QueueInfoItem 接收哪些选项：

```
<QueueInfoItem> must be a member of the list [name, durable,
auto_delete, arguments, pid, owner_pid, exclusive_consumer_pid,
exclusive_consumer_tag, messages_ready, messages_unacknowledged,
messages_uncommitted, messages, acks_uncommitted, consumers,
transactions, memory]. The default is to display name and (number of)
messages.
```

这段内容说明了如果你想要了解更多队列的信息的话，例如名字、消息数目或

者消费者数目，以及内存使用情况，则可以发送以下命令：

```
$ ./rabbitmqctl list_queues name messages consumers memory
Listing queues ...
msg-inbox-logs 0 2 34632
msg-inbox-errors 0 1 34632
all-logs 3 0 43664
...done.
```

正如期望的那样，rabbitmqctl 返回了队列的名称，msg-inbox-logs；消息数目，0；附加的消费者数目，0；以及内存使用情况，43 664 字节。

你可以检查声明队列时使用的属性：

```
$ ./rabbitmqctl list_queues name durable auto_delete
Listing queues ...
msg-inbox-logs true false
msg-inbox-errors true false
all-logs true false
...done.
```

你可以看到队列是可持久化的，同时 auto_delete 属性被设置成 false。

当然，你可以在命令行上放手尝试其他选项。

查看交换器和绑定

现在我们来查看交换器的信息。在这种情况下，获取默认信息的命令如下：

```
$ ./rabbitmqctl list_exchanges
Listing exchanges ...
logs-exchange     topic
amq.rabbitmq.log     topic
amq.match    headers
amq.headers    headers
amq.topic    topic
amq.direct    direct
amq.fanout    fanout
    direct
...done.
```

默认情况下，这个命令返回交换器名称和类型。你会发现若干个交换器早已声明好了，例如 amq.topic、amq.direct 和 amq.fanout。AMQP 规范里对这些交换器有规定。在列表的顶部你可以看到自己定义的 logs-exchanges，类型为 topic。如果你仔细观察结果的底部，就会发现只有一个 direct 类型而没有交换器

名称。这就是我们在第1章提到过的匿名交换器。之后你就会看到，每个队列默认都会绑定到该交换器。

让我们不携带参数运行 rabbitmqctl 命令，看一下 list_exchanges 的参数：

```
$ ./rabbitmqctl
        ...
<ExchangeInfoItem> must be a member of the list [name, type,
durable, auto_delete, arguments]. The default is to display name
and type.
        ...
```

这里的信息是之前用来声明交换器的选项。让我们看看：

```
$ ./rabbitmqctl list_exchanges name type durable auto_delete
logs-exchange    topic    true    false
amq.topic    topic    true    false
amq.direct    direct    true    false
amq.fanout    fanout    true    false
    direct    true    false
```

通过这个命令，你可以知道 logs-exchange 交换器是可持久化的（durable），并且它不会被服务器自动删除。

在检测队列和交换器的信息之后，你自然而然想要看看绑定信息了吧。在命令行键入此命令：

```
$ ./rabbitmqctl list_bindings
Listing bindings ...
    all-logs    all-logs    []
    msg-inbox-errors    msg-inbox-errors    []
    msg-inbox-logs    msg-inbox-logs    []
logs-exchange    all-logs    #    []
logs-exchange    msg-inbox-logs    *.msg-inbox    []
logs-exchange    msg-inbox-errors    error.msg-inbox    []
...done.
```

该命令不接收除 -p（指明 vhost 路径）以外的参数。结果由若干行组成，每行包含了交换器名称、队列名称、路由键和参数。最开始的三行看起来非常特殊，好像缺失了什么……那又是匿名交换器的关系。这几行展示了每个队列使用自己的队列名称作为路由键绑定到匿名交换器上。你也可以看到 logs-exchange（topic 类型交换器）绑定了三个队列。第一个队列 alllogs，通过使用 # 路由键（通配符）进行绑定。另外两个队列则分别使用 *.msg-inbox 和 error.msg-inbox 作为路

由键进行绑定。

我们已经讲述了许多 rabbitmqctl 命令，而且还有更多命令可用（通过不携带参数运行 rabbitmqctl，会看到帮助文档，里面有详细说明）。当我们讨论集群和监控 RabbitMQ 时，你就能了解更多。

3.3.2 理解 RabbitMQ 日志

在之前的章节中，你学到了如何使用 rabbitmqctl 脚本获取服务器信息。那么现在让我们看看你如何检查服务器上真实的情况——正在发生什么类型的事件。RabbitMQ 记录事件有多种原因，例如连接尝试、用户创建及解析请求时的错误。RabbitMQ 有个非常酷的功能是，你可以使用 AMQP 交换器、队列和绑定实时获得所有这些数据，并对此做出反应。如果你想要按照通常的做法的话，也可以在文件系统查看日志文件。让我们先以这种方式进行查看，再构建一个 AMQP 消费者来显示 RabbitMQ 日志；之后你可以调整它并为 RabbitMQ 设置一个监控系统。

在文件系统中读日志

当检查日志时，你需要关心的设置是 LOG_BASE 环境变量。默认值，在 rabbitmq-server 脚本显示如下：

```
LOG_BASE=/var/log/rabbitmq
```

在那个文件夹内 RabbitMQ 会创建两个日志文件：RABBITMQ_NODENAME-sasl.log 和 RABBITMQ_NODENAME.log。其中这里的 RABBITMQ_NODENAME 指的是 _rabbit@localhost_ 或者就是 rabbit，这取决于你如何配置系统。

sasl 日志和另一个日志有什么区别呢？ SASL（System Application Support Libraries，系统应用程序支持库）是库的集合，作为 Erlang-OTP 发行版的一部分。它们帮助开发者在开发 Erlang 应用程序时，提供一系列标准。其中之一是日志记录格式。所以，当 RabbitMQ 记录 Erlang 相关信息时，它会将日志写入 rabbit-sasl.log 文件。举例来说，你可以在这个文件中找到 Erlang 的崩溃报告。这有助于调试无法启动的 RabbitMQ 节点。

现在如果你想看看服务器正在发生的事件的话，则可以 tail -f rabbit.log 文件。你会看到如下这样：

```
=INFO REPORT==== 10-Sep-2010::13:50:58 ===
accepted TCP connection on 0.0.0.0:5672 from 192.168.1.253:44550

=INFO REPORT==== 10-Sep-2010::13:50:58 ===
starting TCP connection <0.29749.52> from 192.168.1.253:44550

=INFO REPORT==== 10-Sep-2010::13:50:58 ===
closing TCP connection <0.29749.52> from 192.168.1.253:44550

=INFO REPORT==== 10-Sep-2010::13:51:08 ===
Rolling persister log to
"/var/lib/rabbitmq/mnesia/rabbit/rabbit_persister.LOG.previous"
```

这段信息对调试你的消费者/生产者很有帮助；你可以看到它们是否连接正常，连接是否突然中断，等等。你也可以找出是否通过一个未经允许的IP地址连接到服务器上。

除了网络流量信息外，从rabbit.log文件你还能看到像对用户、交换器、队列等的操作事件。因此如果某些原因致使AMQP客户端编码请求失败，或者声明队列时有冲突，你都可以在这里找到事件记录。

轮换日志

最后一点内容是关于如何轮换日志文件。首先你需要知道服务器是否启动，它会重新创建日志文件并在旧的文件后面添加一个数字。你会得到像rabbit.log.1这样的文件。如果你想手动轮换日志文件或者通过cronjob来达成的话，则可以使用——你猜得没错——rabbitmqctl。你可以像这样运行：

```
$ ./rabbitmqctl rotate_logs suffix
```

这里的suffix是一个单词，通常是数字，你可以添加到被轮换的日志文件的末尾。你可以尝试这样做：

```
$ ./rabbitmqctl rotate_logs .1
```

你应该在日志文件夹下看到以下文件：

```
$ ls /var/log/rabbitmq
   rabbit@mrhyde-sasl.log
   rabbit@mrhyde-sasl.log.1
   rabbit@mrhyde.log
   rabbit@mrhyde.log.1
```

通过 AMQP 实时访问日志

现在让我们看看如何通过 AMQP 获得实时日志信息。也许当你使用 rabbitmqctl 列出交换器列表时看到了其中一个叫作 amq.rabbitmq.log 的 topic 交换器。RabbitMQ 把日志信息发布到该交换器上，并以严重级别作为路由键——error、warning 和 info。基于从前面章节学到的知识，你可以创建一个消费者来监听日志并做出相应的反应。为了示例起见，这里只将日志内容输出到控制台上。

在编写消费者之前，让我们先做点重构。在 config 文件夹下创建 config.php 文件，并将下面的代码键入其中。

清单 3.2 默认配置文件

```
<?php
define('HOST', 'localhost');
define('PORT', 5672);
define('USER', 'guest');
define('PASS', 'guest');
define('VHOST', '/');
?>
```

从现在开始，我们将假设已将 config.php 文件和 amqp.inc 库包含在内，因此在将来的代码示例中将不再提及。同时，我们也会假设你初始化了连接并且获得了通信信道。记住这些，现在来看看 PHP 消费者示例如下。

清单 3.3 日志监听器

```
list($errors_queue, , ) = $ch->queue_declare();
list($warnings_queue, , ) = $ch->queue_declare();
list($info_queue, , ) = $ch->queue_declare();

$exchange = 'amq.rabbitmq.log';

$ch->queue_bind($errors_queue, $exchange, "error");
$ch->queue_bind($warnings_queue, $exchange, "warning");
$ch->queue_bind($info_queue, $exchange, "info");

$error_callback = function($msg){
  echo 'error: ',  $msg->body, "\n";
  $msg->delivery_info['channel']->basic_ack(
                    $msg->delivery_info['delivery_tag']);
};

$warning_callback = function($msg){
  echo 'warning: ',  $msg->body, "\n";
  $msg->delivery_info['channel']->basic_ack(
```

❶ 声明了三个监听队列

❷ 将队列绑定到日志交换器

❸ 创建回调函数

```
                    $msg->delivery_info['delivery_tag']);
};

$info_callback = function($msg){
  echo 'info: ',  $msg->body, "\n";
  $msg->delivery_info['channel']->basic_ack(
                    $msg->delivery_info['delivery_tag']);
};

$ch->basic_consume($errors_queue, "", false,
                                  false, false, false,
                                  $error_callback);                    ← 准备消
$ch->basic_consume($warnings_queue, "", false,                         ❹ 费者
                                  false, false, false,
                                  $warning_callback);
$ch->basic_consume($info_queue, "", false,
                                  false, false, false,
                                  $info_callback);

while(count($ch->callbacks)) {
  $ch->wait();                                                ← ❺ 等待消息到达
}
```

让我们看看这段脚本做了什么。在❶处你声明了三个队列。如你所见，你并没有提供任何参数给 declare 命令，因此 RabbitMQ 会为队列指定随机的名称。此外，队列对于消费者来说将会是 exclusive 的，因而当脚本停止的时候，也会被自动删除。通过这种方式，你可以在任何时候把消费者附加到 RabbitMQ 上，观测服务器状态，然后从服务器断开，一切都会为你收拾干净。你无须在之后删除队列。你将此命令返回的队列名称分别存储为 $errors_queue、$warnings_queue 和 $info_queue。

在❷处你使用那些参数把队列绑定到 amq.rabbitmq.log 交换器上。你使用不同的路由键将想要的日志信息路由到指定队列上。然后在❸处，你定义了三个回调函数来处理消息。它们基本上做了一样的事情：输出日志内容，并以日志警告作为前缀。你可以修改它们以执行更高级的任务。

然后在❹处你发送了 basic_consume 命令，并设置回调函数来从服务器接收消息。最后在循环处❺，等待发送过来的消息。如果运行该脚本，你应该看到下列日志：

```
info: closing TCP connection <0.25403.2> from 127.0.0.1:54197
info: accepted TCP connection on 0.0.0.0:5672 from 127.0.0.1:54204
```

如果你好奇自己刚才创建的队列名称是什么的话，那就来到另一个终端窗口，并使用 rabbitmqctl 来列出队列列表。你会看到如下所示：

```
Listing queues ...
amq.gen-kkcRbifmFzl4cVI6FLA4fQ==    0
amq.gen-4dngVZQA3QZOUf1obu391w==    0
amq.gen-NeTS98PHQygG3S2ciSzOww==    0
...done.
```

以这种方式创建队列的好处在于你无须担心队列的名称。

就如你在本节中看到的那样，RabbitMQ 为内部状态提供了足够的信息。你有两种方式来检测服务器状态：其中之一是通过传统的文件日志；而另一种是更为先进的 AMQP 交换器，它能给予你充分的灵活性，使得过滤日志变得更加简单。

3.4 修复Rabbit：疑难解答

到目前为止一切都很正常，但是如果你的 Rabbit 难以驯服该怎么办呢？不管你尝试什么方法，它就是不启动，或者就算运行了也不回复你的消息。让我们看看你该如何解决这些问题。

由 badrpc、nodedown 和其他 Erlang 引起的问题

RabbitMQ 出错时返回的稀奇古怪的消息会将 RabbitMQ 新手拒之门外。随着时间的推移，我们所学到的大部分神秘消息均来自于 RabbitMQ 所运行的底层系统 Erlang。我们并不是说 Erlang 是有问题的（远非如此）；我们要说的是像 `badrpc`、`nodedown` 这样的消息是由 Erlang 虚拟机产生的；同时，只需要对 Erlang 的工作机制稍加了解，我们就能轻松解决这些问题。

Erlang cookie

当开始使用 RabbitMQ 时，一些常见的错误是 `badrpc` 和 `nodedown`。这些错误经常在你尝试使用 `rabbitmqctl` 命令的时候发生。取代期望结果的是错误消息的反馈。一开始你会以为 RabbitMQ 没有运行，但执行如下命令就证明之前的想法是错的：

```
$ ps ax | grep rabbit
34373 ?? S 0:01.67 /usr/local/lib/erlang/erts-5.8.5/bin/beam.smp -W
... omitted output ...
dir "/var/lib/rabbitmq/mnesia/rabbit@mrhyde" -noshell -noinput
```

服务器进程正在运行，那么究竟是怎么回事呢？让我们先试着理解 rabbitmqctl 命令的工作原理。rabbitmqctl 会启动 Erlang 节点，并从那里使用 Erlang 分布式系统尝试连接 RabbitMQ 节点。要达成这项工作，你需要两样东西：合适的 Erlang cookie 和合适的节点名称。那么什么是 Erlang cookie 呢？ Erlang 节点通过交换作为秘密令牌的 Erlang cookie 以获得认证。由于你一连接到远程节点后，就能执行命令，因此有必要确保该节点是可信的。Erlang 将令牌存储在名为 .erlang.cookie 的文件。该文件通常位于用户的 home 目录下。你可以执行以下命令看看内容：

```
$ cat ~/.erlang.cookie
```

为了能让 rabbitmqctl 连接 RabbitMQ 节点，因此需要共享相同的 cookie。如果运行 RabbitMQ 和执行 rabbitmqctl 命令的是同一个用户，那么你不会有任何问题。但是在产品环境中，你可能会想要创建 rabbitmq 用户，并以该用户的身份运行服务器。这意味着你必须和 rabbitmq 用户共享 cookie，或者你切换到 rabbitmq 才能成功执行 rabbitmqctl。当我们讨论集群多个 RabbitMQ 服务器时，我们会再次讨论 Erlang cookie 的。

Erlang 节点

关于节点，它们会有什么问题呢？当你启动 Erlang 节点时，你可以给它两个互斥的节点名选项，name 和 sname。节点名可长可短，这也是 sname 中 s（hort）的含义。如果用长名启动节点，则它会像 rabbit@hostname.network.tld 这样。如果你用短名的话，它就会像 rabbit@hostname 这样。后者是启动 RabbitMQ 的默认方式。当你想让 rabbitmqctl 能连接上 RabbitMQ 时，你必须使得这些参数两边都能匹配。让我们实验一下，看看如果将 rabbitmqctl 文件中的 -sname rabbitmqctl$ 修改为 -name rabbitmqctl$ 的话，会发生什么。保存文件并运行以下命令：

```
./rabbitmqctl    list_queues
Listing queues ...
```

你会得到友好的错误消息，像这样：

```
=ERROR REPORT==== 21-Sep-2010::16:01:46 ===
** System running to use fully qualified hostnames **
** Hostname mrhyde is illegal **
Error: unable to connect to node rabbit@mrhyde: nodedown
diagnostics:
```

```
- nodes and their ports on mrhyde: [{rabbit,54174}...]
- current node: 'rabbitmqctl1027@mrhyde.network.tld'
- current node home dir: /Users/mrhyde
- current node cookie hash: oNANSQ6MP0092ATN9U7Hcc==
```

将选项更改回读取 sname 后，一切正常。之后当我们讨论集群时，我们将会调整该选项。

Mnesia 和主机名

接下来我们来讨论 Mnesia。作为 Erlang 数据库，它早在 NoSQL 流行之前就已经存在了。RabbitMQ 使用 Mnesia 存储队列、交换器、绑定等信息。RabbitMQ 启动时做的一件事就是启动 Mnesia 数据库。由于这是服务器正常运行的关键一步，因此如果 Mnesia 启动失败，那么 RabbitMQ 也会跟着失败。

导致 Mnesia 启动失败的原因大致有二。第一个也是最常见的是 MNESIA_BASE 目录的权限问题。运行 RabbitMQ 服务器的用户需要对该文件夹的写权限。另一个常见的问题是你看到这样的错误消息：

```
starting database                  ...{"init terminating in
do_boot",{{nocatch,{error,
    {cannot_start_application,rabbit,
        {{timeout_waiting_for_tables [...]
```

可以从消息中看到，Mnesia 读取表格失败。如果主机名更改了，或是服务器运行在集群模式下，无法在启动的时候连接到其他节点，这些都会导致失败。为什么需要关注主机名呢？ Mnesia 会基于机器的主机名创建数据库 schema。如果你列出 MNESIA_BASE 文件夹的内容，就会看到其中一个文件夹是 rabbit@hostname。如果由于网络重新配置的原因，主机名修改了，那么 Mnesia 就无法载入旧的 schema。同时请谨记 RabbitMQ 使用 rabbit 这个单词作为节点名。如果你使用 Erlang sname 选项更改了它，Mnesia 又会遇到同样的问题。

基于相同的原因，你会发现如果你重命名了 rabbit@hostname 文件夹，那么 Mnesia 将无法找到旧的数据库文件。它会重新创建 rabbit@hostname 文件夹并启动数据库。记住：你仍然可以在你重命名的文件夹下找到旧的数据库文件。

Mnesia 很值得用一本书来讲述它。所以如果你想要深入了解，就可以阅读用户指南 http://www.erlang.org/doc/apps/mnesia/users_guide.html。

Erlang 故障排除技巧

让我们做一个简单的 Erlang 练习来结束本节。它会帮助你理解我们之前所讨论的内容。你将 Erlang 节点连接到运行中的 RabbitMQ 服务器。你可以用这一知识监控运行的服务器，远程执行 Erlang 函数，等等。

假设你用短名启动 RabbitMQ，运行以下命令：

```
$ erl -sname test
```

依赖于你的主机名和 Erlang 版本，你应该会看到以下内容：

```
Erlang R13B04 (erts-5.7.5) [source] [64-bit] [smp:2:2] ...
Eshell V5.7.5 (abort with ^G)
(test@mrhyde)1>
```

这意味着你用 test 作为节点名启动了 Erlang 节点。你看到的是 Erlang Read Eval Print Loop，简称 REPL。你可以在此输入命令，执行 Erlang 代码等。让我们找出节点名称：

```
(test@mrhyde)1> node().
test@mrhyde
```

在 REPL 中，你输入 node() 并得到结果 test@mrhyde。你的 Erlang 节点将以这个名字被其他 Erlang 应用程序知晓。举例来说，如果你启动 Mnesia 数据库的话，这就会被用作 schema 文件夹的名字。

现在让我们看看在你的机器上还运行着哪些节点。

```
(test@mrhyde)2> net_adm:names().
{ok,[{"rabbit",59106},{"test",59127}]}
```

你从 net_adm 模块调用了 names 函数。如你所见，RabbitMQ 正运行在你的机器上。它使用 rabbit 作为节点名称并使用 59106 端口。等等……那并不是你从 AMQP 客户端用来连接 RabbitMQ 的端口。这是怎么回事呢？这就需要另外一位角色登场了：Erlang Port Mapper Daemon（epmd）。当你启动一个分布式 Erlang 节点时，它会用 epmd 进程进行注册，提供 OS 内核分配的地址和端口。之后当另一个 Erlang 节点启动时，它也会做同样的事情。最后，如果后者想要连接第一个节点的话，它会查阅 epmd 以获得节点地址。通过这种方式，你就不需要自己来追踪信息了。然后，那个端口号（示例中的 59106）是由 OS 分配给运行 RabbitMQ 的 Erlang VM 的。

现在你明白如何查找 RabbitMQ 节点了，让我们尝试建立到它的连接：

```
(test@mrhyde)3> net_adm:ping('rabbit@mrhyde').
pong
```

你使用 ping 函数尝试对其他节点的连接，参数为你想要连接的 'node@hostname'。如果应答是 pong 的话，则表明连接成功了；如果应答是 pang，那意味着你无法连接到该节点。记住为了使一切工作正常，你必须共享同样的 Erlang cookie。

让我们确认你连上了 rabbit 节点：

```
(test@mrhyde)4> nodes().
[rabbit@mrhyde]
```

执行 nodes() 函数会展示——Erlang 中方括号为界的列表——你连接上的节点列表。

最后，让我们在远程 rabbit 节点上执行一个函数：

```
(test@mrhyde)8> rpc:call('rabbit@mrhyde',
                         erlang,
                         system_info,
                         [process_count]).
106
```

这里你使用了 Erlang 的 rpc 模块在 rabbit@mrhyde 节点调用了一个函数。你使用 erlang: system_info 函数以获得在该节点上 Erlang 进程的数目。你可以使用该函数监控 RabbitMQ 系统的健康状况。如果 process_count 超过了 process_limit 的限制，你的服务器就会崩溃。这不太可能发生——有些机器的限制是 1048576——但将这些铭记于心总是有用的。

通过使用 rpc:call，同时提供节点、模块、函数和参数作为入参，你可以在远程 rabbit 上执行其他函数以获得不同的信息。最后一个示例，让我们搜集一下远程 rabbit 节点上运行着的 Mnesia 系统的信息：

```
rpc:call('rabbit@mrhyde', mnesia, info, []).
```

REPL 会打印关于 Mnesia 的若干行信息，比如 RabbitMQ 创建的表格、内存使用情况等。

最后，执行 q 函数退出 Erlang REPL。这时，你应该回到了终端命令提示符：

```
(test@mrhyde)11> q().
ok
```

Erlang 值得用一本书来讲述。不过先把这个想法放一边，你可以使用这里描述的技巧来解决 RabbitMQ 的安装问题。举个例子，如果你使用和 rabbitmqctl 脚本相同的参数启动 Erlang VM 的话，就应该可以连接到 rabbit 节点。如果不是这样的话，你也能找出机器的主机名。然后你能列出注册在 epmd 守护进程的节点名字等。有了这些简单的工具，你可以随时随地排查故障并且监控你的 RabbitMQ 服务器。

3.5 总结

在本章中，我们介绍了许多实用技术，能给日常 RabbitMQ 工作带来帮助。你看到了如何执行服务器管理任务，比如使用 RabbitMQ 权限系统来添加和删除用户。我们介绍了 RabbitMQ 配置文件并说明了如何使用 rabbitmqctl 命令——它是服务器日常管理的“瑞士军刀”。然后你看到如何获取服务器的数据统计，以查看你创建的队列、交换器和它们之间的关系。最后一点同样重要，我们讲解了 RabbitMQ 可能向你一次又一次抛出的奇怪的 Erlang 错误。由于 Erlang 是服务器结构的基础，因此你若对该语言有所了解的话，就能执行更高级的管理任务，同时也给了你坚实的 RabbitMQ 原理和 Erlang 语法基础。有了这些知识，让我们进入第 4 章去看看一些真实的例子：怎样用消息通信武装你的应用程序。

4 解决Rabbit相关问题：编码与模式

本章要点

- 面向消息通信来设计应用程序
- 消息通信模式
- 发后即忘模型
- 用RabbitMQ实现RPC

至此你已经知道了如何安装、配置，并在产品环境中运行 Rabbit。现在是时候让我们开始写点代码了。首先，当你将消息通信集成到应用程序时，你需要理解你正在尝试解决的问题。像探索 RabbitMQ 的大多数人一样，我们并不是在寻找一个消息队列，而是解决解耦问题方法。你如何将一个耗时的任务从触发它的应用程序中移出（因该应用程序可以继续服务其他请求）？同样,你如何整合用不同语言（譬如 PHP 和 Erlang），编写的应用程序，使得它们运作起来像单个系统？这看起来像两个不同的问题，但却有着共同的本质：解耦请求和操作。或者换种说法，这两个问题均需要从同步编程模型转向异步编程模型。

通常情况下，当程序员听到异步编程时，他们要么吓跑了，要么在想“酷。就像 Node.js 那样对吧？”有时则兼而有之。通常，异步编程解决方法的问题在于它

们是全有或全无的方案。要么重写所有代码以至于没有一处是同步的，要么就是在浪费时间。RabbitMQ 为你提供了不同的方法。它允许你发送请求并在另一处进行处理，这样同步程序可以返回继续运行。当开始使用消息通信时，你可以获得大部分纯异步编程的益处，又不用将现有的代码和知识抛开。在本章中，我们将向你展示 Rabbit 世界中异步编码的含义。特别是，我们将向你展示如何使用 Rabbit 去解决一系列现实世界中的问题。这些问题包括从图片上传（并行处理）、告警（通知），到使用 RabbitMQ 轻而易举实现分布式远程过程调用（API）。我们会从讲解一些基础的消息通信方式开始，然后逐步深入到编码。让我们马上来解耦吧！

4.1 解耦风雨路：谁将我们推向消息通信

你花费了所有精力。你编写了最新的、最伟大的 Web 应用程序（安排吉娃娃遛狗程序），并决定最快的方式就是接受网络订单，并把它们直接装填到数据库去。听起来很有道理。把一条很小的记录写到数据库里需要花费多少时间呢？更不用说编写起来有多简单了。问题在于，当你走向全国，每小时调度安排 100 000 条吉娃娃遛狗记录时，该怎么办？或者更好的是，你决定要将数据存储到两个地方（将这些请求进行归档）。猜猜看该怎么办？是时候剥离那些精心调试的代码了。应用程序和存储之间直接耦合通常就是导致之后代码被淘汰和取代的原因。这正是消息通信能够帮助你的地方。

4.1.1 异步状态思维（分离请求和动作）

你多久会做一次同步动作？如果你订了一份比萨，你会一直等到它送上门来而不做其他事儿吗？当然不会了。你看看电视或者读读书，或者和她甜言蜜语一会儿。你极少会把自己的生活搁置起来去等待请求的回应。你一心多用，所以你的生活才能丰富多彩。你的应用程序需要同样的方法。

为何你会首先将应用程序设计成同步的呢？大多数情况下是因为你思考着整个任务，而非着眼于构成整体的更小的任务。你脑袋里想着：“我的应用程序需要安排一个吉娃娃遛狗预约。”相反，现实情况是你的应用程序需要接收一个安排请求；然后它需要将请求存储到数据库中；之后它需要向就近的遛狗师发送提示；最终需要让客户知道预约成功了。即便你将应用程序打造成多线程的，仍然会严重限制接收订单的速率。因为每个线程都不得不等待记录的存储以及遛狗师收到通知。而你

应该将这四个步骤视为两个分隔的应用：其中一个专门接收请求，另一个专门处理请求。从 Gregor Hohpe 那里窃取的灵感来做个伟大的类比，我们称之为 Coffee Bean 模型（Coffee Bean & Tea Leaf 是加州的一家连锁咖啡馆）。

当下了个拿铁订单时，你并不会一直等在收银台。相反，Coffee Bean 会将接收订单的操作和订单准备的操作分开。接单员会收集你的请求（和你的钱），并告诉咖啡调理师你订的是什么。然后你等待订单完成，同时接单员可以腾出时间接其他订单。当中最重要的部分是收你的钱，以及将订单处理和订单接收分开，Coffee Bean 已经最大化了每分钟所能接收的订单数。同样地，如果等待制作的咖啡积压太多的话，他们会增加更多的咖啡调理师以减少订单积压，而无须更改接单员的数量。通过解耦处理过程（分离请求和操作），他们以同样数量的工作人员完成了更多的工作，同时工作又很容易按需扩展。消息通信可以为你的应用程序做同样的事情。

所以让我们用解耦的思想分析一下你的吉娃娃应用程序。图 4.1 展现了完成遛狗订单的步骤。

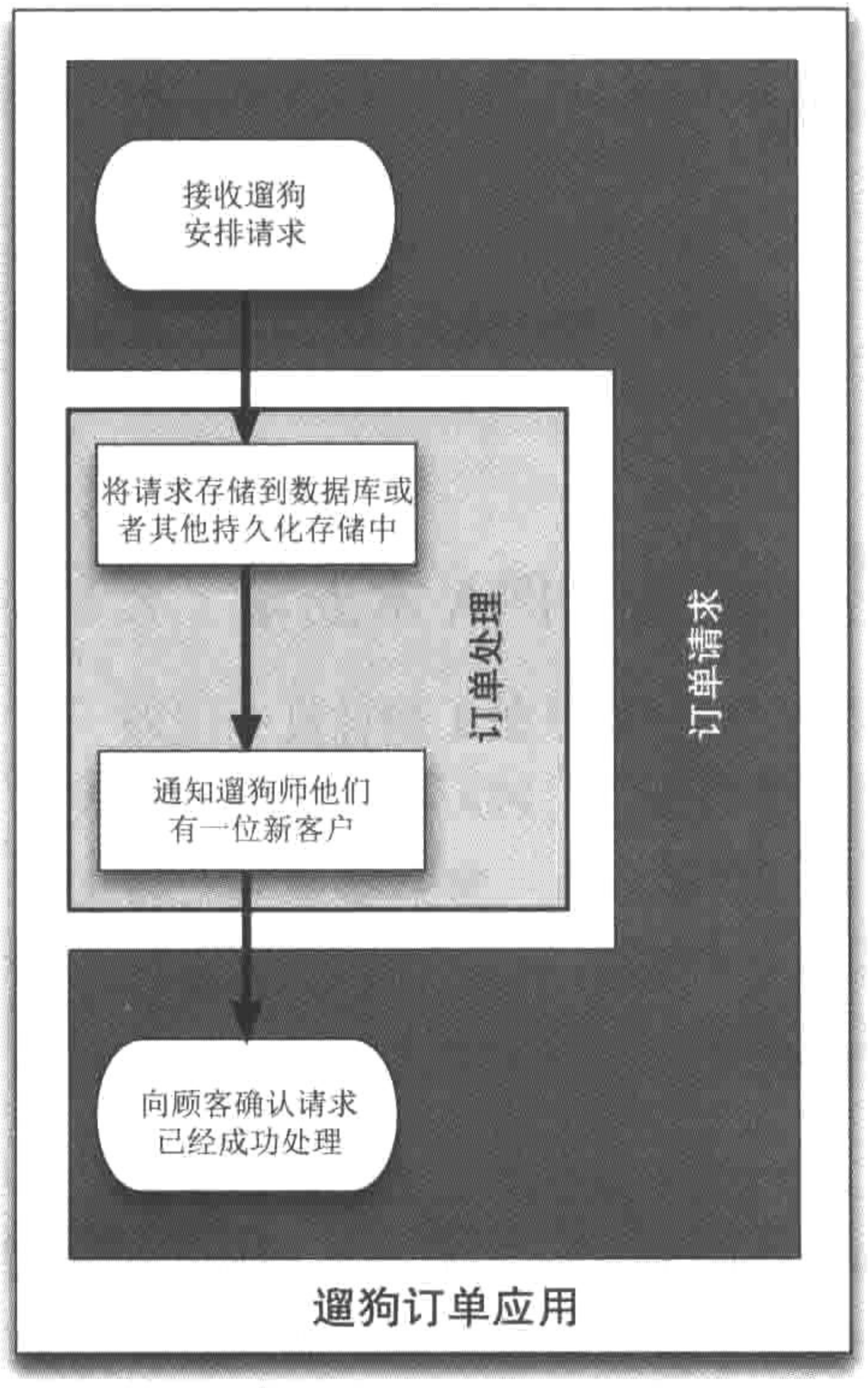

图 4.1 完成一笔遛狗订单的步骤

如果你想要为自己的应用程序增添可扩展性和灵活性的话，就需要将其切分成两个不同的应用：dog_walk_order 和 dog_walk_process，如图 4.2 所示。dog_walk_order 运行在互联网上并接收网络请求以安排遛狗。当收到请求时，dog_walk_order 创建一条新的 AMQP 消息并发送到 Rabbit 上的 `chihuahua_scheduling` 交换器。之后 dog_walk_order 将暂时搁置顾客 A 并继续接收其他请求。同时，dog_walk_process 监听 Rabbit 队列并接收到包含顾客 A 的预约订单的消息。然后它就为订单创建所需的数据库记录，并发送一条文本消息给主遛狗师 Gustav。一旦文本消息发送成功，dog_walk_process 会发送一条顾客 A 请求成功处理的消息回给 dog_walk_order。就在 dog_walk_process 处理顾客 A 请求的那段时间里，dog_walk_order 又接收 100 条新的遛狗请求。如果它们仍然处于同一个应用中的话，那么在整个时间段里就只能接收一条遛狗请求。

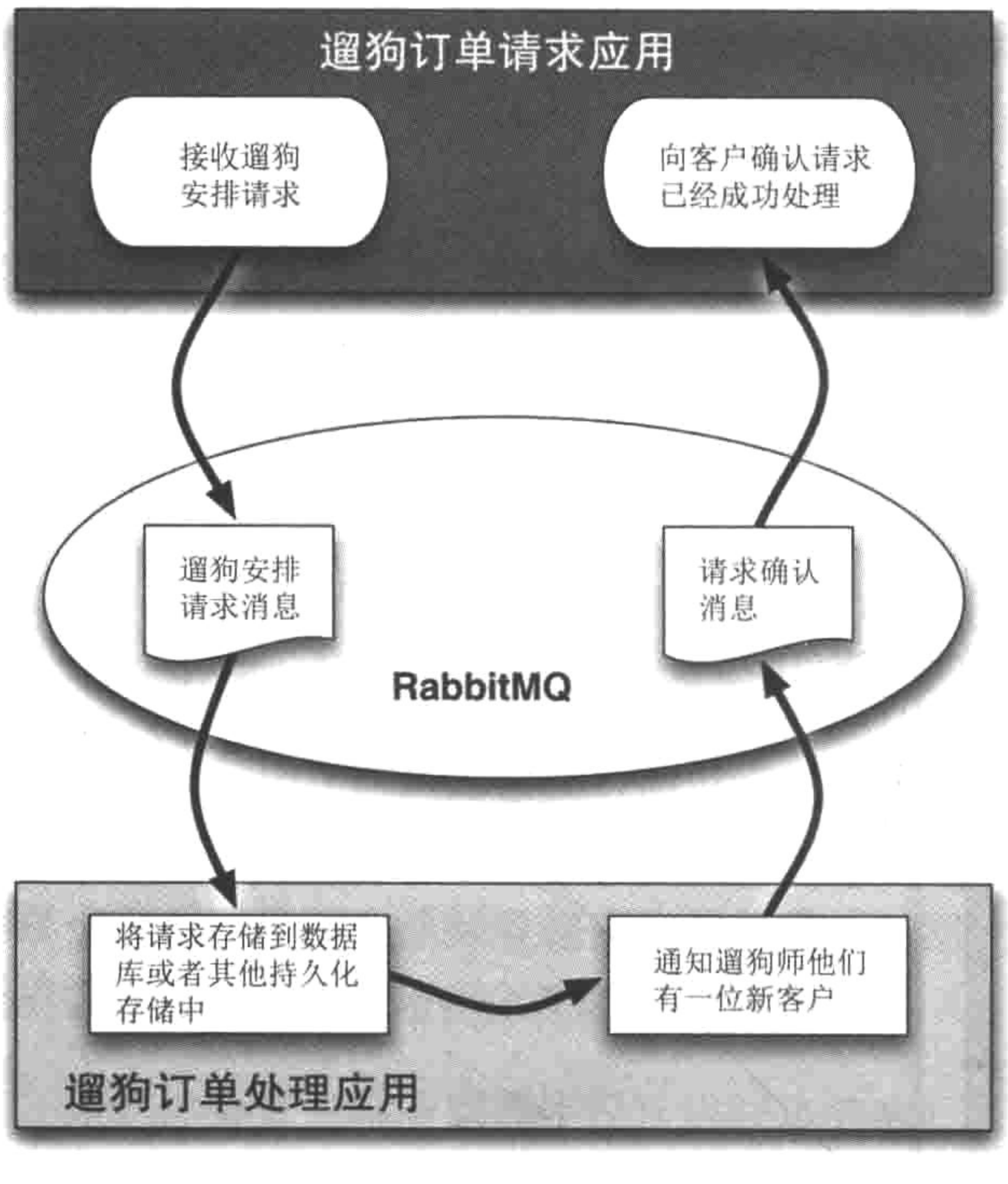

图 4.2 将遛狗程序分离成两个应用

通过在应用中紧密耦合的两部分中间使用 RabbitMQ，之前每次只能处理一个请求而现在可以持续接收请求了。同时你也开辟了另一片新天地。如果负载过重，以至于一台订单处理服务器不够用了该怎么办呢？

4.1.2 提供扩展性：没有负载均衡器的世界

使用消息通信最大的好处之一是，为应用增加处理能力变得简单。假设你刚刚将业务扩展到了日本，现在每秒接收 1 000 000 条遛狗订单。虽然前端订单接收器能跟得上负载并且绰绰有余，但是订单处理器却在垂死挣扎。虽然接收客户订单已经不会再耽搁你接收其他订单了，但是那些客户受够了等待预约的确认。你需要更多的订单处理器。就像在 Coffee Bean 那样，你可以添加更多的咖啡调理师。在该示例中，你增加了额外的 dog_walk_proces 服务器并将它们附加到接受订单的队列上。像变戏法那样，无须更改一行代码，你就通过增加 10 台新的 dog_walk_process 服务器增加了 10 倍的处理能力。最美妙的是，RabbitMQ 会将请求在处理服务器间平均地分发。这归因于我们在 2.2 节讨论过的自动轮询（round-robin）行为模式。这里根本不需要昂贵的负载均衡器。

对任何组织，而不仅仅是金钱意识强烈的初创公司，这都很重要。负载均衡硬件非常昂贵。这也意味着能在多少地方部署它来解耦和扩展应用是受到限制的。如果你使用 AMQP 和 Rabbit 来替代，那么可以在任何想要的地方免费添加解耦和负载均衡。更不用提，你可以实现更复杂的路由方案，譬如将一条消息发送到至少一个目的地，或是轮询负载均衡。负载均衡器总是需要在前端一个固定地点分配来自网络的请求；但是如果你能很好地利用消息通信，那么你就能在防火墙内减少对负载均衡器的依赖，同时极大地增加应用程序的可解耦处。解耦了的应用才是可扩展的。

4.1.3 零成本 API：语言不应成为枷锁

我们一直忽略了使用 AMQP 来解耦应用程序的最大好处：免费的 API。当今所有人都在讨论网络 API，它允许你将应用程序功能集成到其他应用程序当中去。通常来讲，这需要花费一点点努力，因为你最终需要写一大段代码来将传入的 HTTP 请求转化为应用程序中的函数调用。如果你使用 AMQP 来连接应用程序的各个部分，那么实际上你将无须额外的工作就能获得一套 API——这套 API 使用消息通信。

假设你扩展了吉娃娃遛狗，加入了洗狗业务。你有两个新的应用程序来支持这项新业务：dog_wash_request 和 dog_wash_process。然后你有了个绝妙的想法：为每次洗狗赠送一次遛狗服务。由于洗狗和遛狗预约应用程序都使用 AMQP，你所需要做的是更新 dog_wash_request 产生一条额外的包含遛狗预约信息的 AMQP 消息。这

意味着你不需要重写预约代码，也无须复制洗狗应用的代码。同样重要的是，遛狗应用和洗狗应用不需要使用同一门语言编写。

当你编写遛狗应用时，你可能选择 Erlang 作为工作的最佳语言。但在之后的几个月里，你发现自己非常喜欢使用 Clojure 来构建高并发应用程序。因此你用 Clojure 来编写洗狗应用程序。如果使用 Erlang 内建的通信协议来连接 dog_walk_request 和 dog_walk_process 的话，那么让洗狗应用程序和 dog_walk_process 进行通信将是非常困难的，因为它并不是用 Erlang 编写的。但是由于 AMQP 是语言无关的，并且拥有数十种语言的本地语言绑定，因此你可以在 Rabbit 上轻松连接 Clojure 请求接收器和 Erlang 请求处理器。使用 AMQP 连接应用程序给了你足够的灵活性来为工作的每一部分选择正确的语言，并且即使之后改变了主意也能连接用完全不同语言编写的应用程序。你能以想要的任何方式来连接基础架构的任何或者所有部分，RabbitMQ 让这一切变得简单。

所以，首先你要做的是问问你自己，如何将你的应用程序进行切分？或者说，应用程序的哪部分是订单接收者，哪部分是订单处理者？记住这些，让我们深入到一些现实世界的示例当中去，使用 Rabbit 和消息通信解决实际问题和回答这些疑问。

4.2 发后即忘模型

当我们查看消息通信能够解决的问题类型时，消息通信适用的主要领域之一是发后即忘的处理模式。不管是将联系方式添加到邮件列表中去，还是将 1000 张图片转换成缩略图，你关心的是这些任务将会完成，但无须实时完成。事实上，你通常想要避免阻塞触发任务的用户。我们描述了作为发后即忘的三种类型的任务：你创建了任务，放置到交换器上，并让你的应用程序返回继续工作。根据你的需求，你可能甚至都不需要通知用户任务已经完成。

匹配该模式的两种一般类型的任务：

- 批处理（batch processing）——针对大型数据集合的工作或者转换。这种类型的任务可以构建为单一的任务请求，或者多个任务对数据集合的独立部分进行操作。
- 通知（notifications）——对发生事件的描述。内容可以是消息的日志，也可以是真实的报告通知给另一个程序或者是管理员。

我们将向你展示两个不同的现实世界中发后即忘的例子。这两个例子符合我们之前提到的两种类别。第一个是告警框架，它允许基础架构中的应用生成管理员告警通知。你不需要担心它们需要发送到哪里，以及如何到达。第二个例子是关于批处理绝佳的示例：将单张图片上传并将其转换成众多图片尺寸和格式。当本节结束的时候，你将学会 RabbitMQ 编程的最基本类型：用无须应答的消息来触发工作。让我们开始生成些告警信息吧！

4.2.1 发送告警

不管你编写的是何种类型的应用程序，在发生错误时获得通知是非常必要的。典型的情况下，你可能运行像 Nagios 这样的服务监控程序。它会在应用程序宕机或者其依赖的服务不可用时通知你。但是，当应用程序正经历来自同一 IP、非正常数量的用户登录请求时，你是否能收到通知呢？或者也许你允许客户在他们的数据发生不寻常事件时收到通知？你需要的是为应用程序生成告警信息，但是这开辟了一套全新的问题，且会为应用程序带来不小的复杂度。什么样的事件需要告警？更重要的是，你如何进行告警？ SMS？ IM？不管你如何对应用进行切分，你正打算加入的大量代码深藏着众多 bug。举例来说，当 SMS 网关宕机了会发生什么？所有需要告警的 Web 应用程序需要错误处理代码以应对 SMS 服务器不可用。

别担心，RabbitMQ 正赶来拯救你。本质上你的 Web 应用程序唯一要做的是生成告警内容。它们无须知道告警信息应该去到哪里，怎么到达那里，或者当告警信息投递失败该怎么办。所有你需要做的是写一个新的告警服务器应用程序，通过 Rabbit 接收告警消息，然后在适当的时候增强 Web 应用以发布告警消息。

你怎样设计这个新的告警框架？特别是你应该使用什么 AMQP 交换器类型？你应该使用 fanout 交换器，然后为每种告警传输类型（IM、Twitter、SMS 等）创建队列。好处是 Web 应用不需要知道告警信息是如何投递到最终接收者的。它只是发布消息，然后继续工作。坏处是每个告警传输者都会得到一份拷贝，所以每次告警发生的时候，你都会被 IM、文本消息、Twitter 直通消息淹没。

另一种更好的组织告警系统的方式是为其创建三种严重级别告警：info、warning 和 critical。但是如果使用 fanout 类型交换器的话，则任何发布的告警都会发送到所有这三个严重级别队列上。作为替代方案，你可以创建 direct 类型交换

器。它允许应用程序为告警消息贴上严重等级的标签，用来作为路由键。不过，如果你选择 topic 类型交换器会怎样呢？ topic 交换器使得你可以为消息创建灵活的标签，从而将它们引导到多个队列。这些队列可以提供你所需的服务（而非 fanout 交换器那样）。如果你为告警框架采用 topic 交换器，你就不会受限于每个 alert 只有一种严重级别。事实上，你将可以为消息贴上严重级别标记和类型标记。举例来说，假设有个叫 Joe Don 的黑客为了在你的遛狗预约里匹配数据，正以每秒 10 000 次请求的速度攻击你的数据统计服务器。在你的组织里，你需要将警告发送给基础设施管理部门（他们会接收所有标记为 `critical` 的告警信息）和你的 API 开发团队（他们会接收所有标记为 `rate_limiting` 的告警信息）。既然你已经为告警框架选择了 topic 交换器，应用程序就可以为像这样卑劣行径的告警打上 `critical.rate_limiting` 标记。你看！ RabbitMQ 自动将这条告警信息路由到了 `critical` 队列和 `rate_limiting` 队列去了。这是因为你创建的交换器绑定规则：`critical.*` 和 `*.rate_limiting`。图 4.3 展示了告警系统的工作流程。

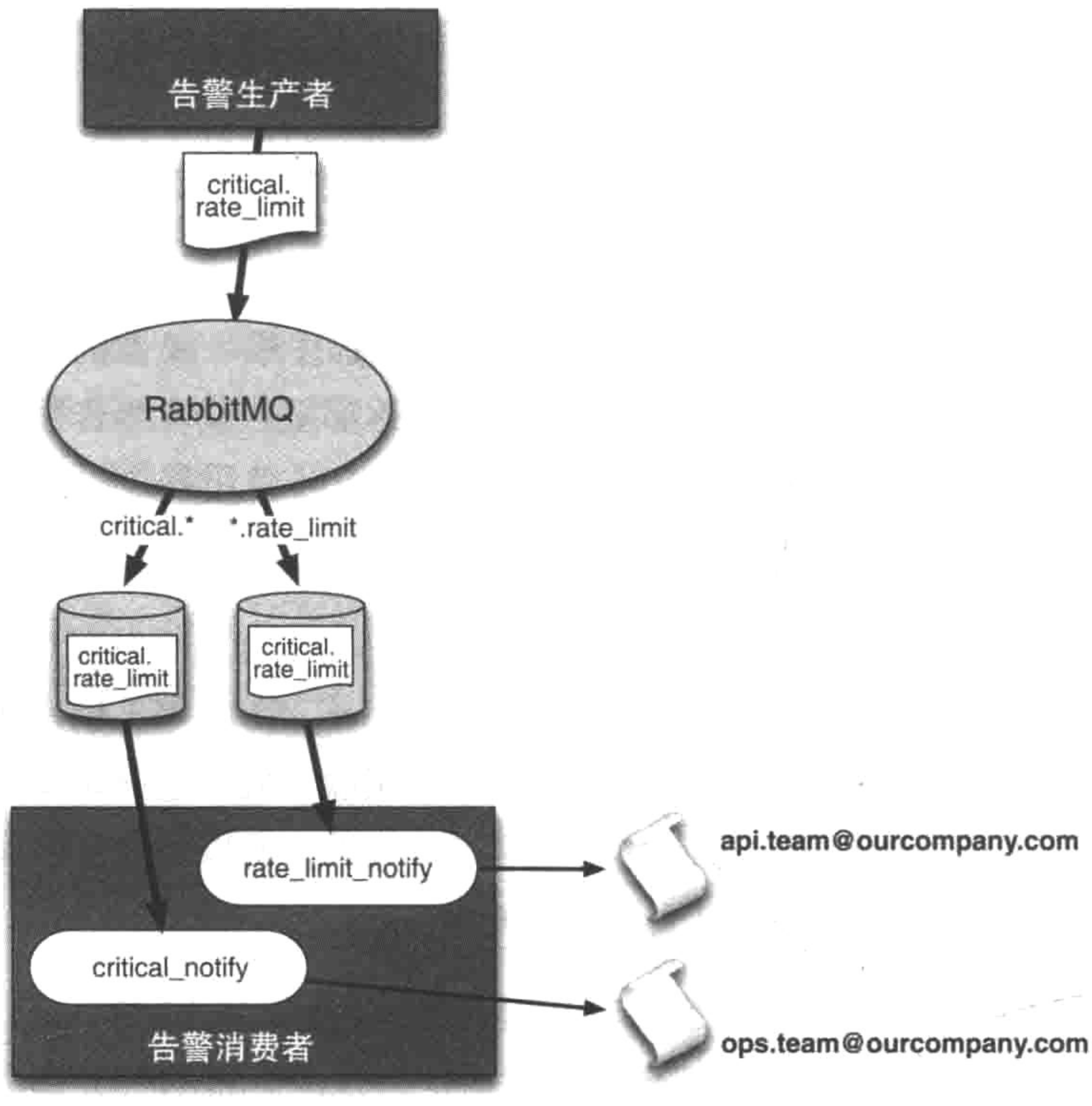

图 4.3 告警系统流程

为了构造这个告警框架，你需要在第 2 章 Hello World 例子中安装的 Pika 库。如果你跳过了那部分的话，这里有 Pika 快速安装步骤（假设你也没有安装 easy_install）：

```
$ wget http://peak.telecommunity.com/dist/ez_setup.py
...
  (25.9 KB/s) - ez_setup.py saved [10285/10285]
$ python ez_setup.py
...
Installed /Library/Python/2.6/site-packages/setuptools-0.6...
$ easy_install pika
...
Installed /Library/Python/2.6/site-packages/pika-0.9.6-py2.6.egg
Processing dependencies for pika
Finished processing dependencies for pika
```

下一步需要为应用程序设置一套 RabbitMQ 用户名和密码来发布和接收告警消息。让我们设置用户名为 alert_user 并且密码为 alertme。同时也给 alert_user 用户授予在默认 vhost “/” 上的 read/write/configure 权限。

从 RabbitMQ 安装目录下的 ./sbin 目录，运行以下命令：

```
$ ./rabbitmqctl add_user alert_user alertme
Creating user "alert_user" ...
...done.
$ ./rabbitmqctl set_permissions alert_user ".*" ".*" ".*"
Setting permissions for user "alert_user" in vhost "/" ...
...done.
```

设置好之后，你可以准备告警系统的最重要部分了：AMQP 消费者会接收告警消息并把它们传输到目的地。创建 alert_consumer.py 文件，并将下列代码复制到文件内。

清单 4.1 连接到代理

```
import json, smtplib
import pika

if __name__ == "__main__":
    AMQP_SERVER = "localhost"
```

代理设置

```
AMQP_USER = "alert_user"
AMQP_PASS = "alertme"
AMQP_VHOST = "/"
AMQP_EXCHANGE = "alerts"

creds_broker = pika.PlainCredentials(AMQP_USER, AMQP_PASS)
conn_params = pika.ConnectionParameters(AMQP_SERVER,
                                        virtual_host = AMQP_VHOST,
                                        credentials = creds_broker)
conn_broker = pika.BlockingConnection(conn_params)
channel = conn_broker.channel()
```

建立到代理的连接

这段代码首先做的是包含了用于创建消费者的库，并告诉 Python 程序的主体位置（`if __name__ =="__main__":`）。下一步，你进行设置并成功连接到代理服务器（用户名、密码、虚拟主机等）。该设置假定 RabbitMQ 运行在你的开发工作站上，同时使用你刚创建的用户名和密码。为简单起见，使用默认虚拟主机“/”，并在其上创建 `alerts` 交换器。让我们真正行动起来：

```
channel.exchange_declare( exchange=AMQP_EXCHANGE,
                          type="topic",
                          auto_delete=False)
```

通过将参数 `type="topic"` 传递给 `channel.exchange_declare`，你声明了 topic 类型的 `alerts` 交换器。传入的 `auto_delete` 参数意味着当最后一个消费者断开连接后交换器仍然会存在。

记得我们讨论过告警的两种打标模式：

- 用来为告警消息标记严重级别（即 `critical`）
- 用来为告警消息标记特定的类型，例如 `rate_limiting`

为了让告警消息正确到达队列，你需要创建绑定来实现这些规则。对该示例来说，让我们创建一个绑定，将所有标记以 `critical.` 起始的消息路由到 `critical` 队列。让我们创建另一个绑定，将所有标记以 `.rate_limit` 结尾的消息路由到 `rate_limit` 队列。继续创建 `critical` 队列和 `rate_limit` 队列，并绑定它们，就像下面展示的那样。

清单 4.2 为告警 topic 声明并绑定队列和交换器

```
channel.queue_declare(queue="critical", auto_delete=False)
channel.queue_bind(queue="critical",
                   exchange="alerts",
                   routing_key="critical.*")

channel.queue_declare(queue="rate_limit", auto_delete=False)
channel.queue_bind(queue="rate_limit",
                   exchange="alerts",
                   routing_key="*.rate_limit")
```

critical队列和critical.* topic绑定

rate_limit 队列和*.rate_limit topic 绑定

你会注意到为 critical 创建的告警绑定的规则是 `critical.*` 而非 `critical*`。这是因为 RabbitMQ 使用“.”作为标记中不同部分的分隔符。如果你用 `critical*` 作为规则的话，那么只有那些实实在在标以 `critical*` 的标记才能匹配。其实你想要匹配的是 `critical.mywebapp`、`critical.rate_limit`，或者任何以 `critical.` 起始的标记。因此，绑定规则应为 `critical.*`。当使用 topic 交换器时，仔细设计标记模式很重要，使用“.”来分隔你想要分别匹配标记的各个部分。

你本可以将 `durable=True` 传入队列声明和绑定，这样就能确保它们幸免于 RabbitMQ 重启。由于重启消费者会自动创建交换器、队列和绑定，因此你不必担心告警系统的持久化。另一个你不关心队列持久化的原因是你同样不想持久化消息。你的系统可能会处理大量的告警信息，所以你想要确保最高的性能，同时不使用持久化消息通信。因为那会将消息存储到相对缓慢的硬盘上。

你也许会好奇：“我们有交换器、队列和绑定……我们在哪里将消息转换为真实的告警呢？”你通过设置消费者订阅并启动监听循环来实现这一点，如下所示。

清单 4.3 将告警附加到处理器上

```
channel.basic_consume( critical_notify,
                       queue="critical",
                       no_ack=False,
                       consumer_tag="critical")

channel.basic_consume( rate_limit_notify,
                       queue="rate_limit",
                       no_ack=False,
                       consumer_tag="rate_limit")

print "Ready for alerts!"
channel.start_consuming()
```

让我们来剖析 `channel.basic_consume` 每个参数的含义：

- critical_notify是回调函数。你订阅了critical队列，当消息接收到以后，就会调用该方法。当Pika库在该订阅上收到消息后会调用critical_notify函数，传入消息的信道、消息头、消息体和消息方法。
- queue="critical"明确指定了你想要从哪个队列获取消息。
- no_ack=False告诉RabbitMQ你想要显示确认收到的消息。这会让Rabbit暂停从队列发送新的消息过来，直到你收到的最后一条消息处理完成并发送确认消息为止。
- consumer_tag是一个标识符，它会在你用channel = conn_broker.channel()创建的AMQP信道上唯一标志此订阅。当你想要取消订阅时，你需要将该消费者标记传给RabbitMQ。

一旦你建立了消费者订阅，则只需调用channel.start_consuming()函数来启动消费者监听消息。你可能注意到了你为订阅指定的回调函数（critical_notify和rate_limit_notify）尚未定义。让我们继续，在下列清单中明确其中一个回调函数。

清单4.4 critical告警处理器

```
def critical_notify(channel, method, header, body):
    """Sends CRITICAL alerts to administrators via e-mail."""

    EMAIL_RECIPS = ["ops.team@ourcompany.com",]

    message = json.loads(body)

    send_mail(EMAIL_RECIPS, "CRITICAL ALERT", message)
    print ("Sent alert via e-mail! Alert Text: %s  " + \
           "Recipients: %s") % (str(message), str(EMAIL_RECIPS))

    channel.basic_ack(delivery_tag=method.delivery_tag)
```

从JSON格式解码消息

将email传输到SMTP服务器

确认消息

当消费者回调函数被调用时，Pika会传入消息相关的四个参数：

- channel——和Rabbit通信的信道对象。如果同时有多个信道打开的话，这里会是其中关联到接收到消息的订阅的信道。
- method——一个方法帧对象。携带了关联订阅的消费者标记以及投递标记。
- header——表示AMQP消息头的对象。它们会携带可选的消息元数据。
- body——实际消息内容。

在critical_notify中，首先要做的事情是检查content_type头。由于告

警信息是 JSON 编码的，因此你要检查内容类型以确保是“application/json”。虽然 content_type 是可选的，但是当你想要在生产者和消费者的通信中编码信息的话，它还是很重要的。在验证了 content type 后，你就可以把消息体从 JSON 解码为文本，然后构造一封包含告警内容的邮件给 Ops Team（ops.team@ourcompany.com）。一旦邮件告警成功发送后，你会发送一个确认消息给 RabbitMQ，说明你已经接收到了消息。确认消息非常重要，因为 RabbitMQ 会直到你确认最后一条收到的消息后，才会发送新的消息给你。通过将消息确认放置在最后一步，你就能确保如果消费者崩溃了的话，RabbitMQ 也会将该消息分发给另一个消费者。

通过对消费者所有部分的解释，让我们看看将所有这些放在一块儿的完整代码，具体如下所示。

清单 4.5 完整的告警消费者代码——alert_consumer.py

```
import json, smtplib
import pika

def send_mail(recipients, subject, message):
    """E-mail generator for received alerts."""
    headers = ("From: %s\r\nTo: \r\nDate: \r\n" + \
               "Subject: %s\r\n\r\n") % ("alerts@ourcompany.com",
                                         subject)

    smtp_server = smtplib.SMTP()
    smtp_server.connect("mail.ourcompany.com", 25)
    smtp_server.sendmail("alerts@ourcompany.com",
                         recipients,
                         headers + str(message))
    smtp_server.close()

def critical_notify(channel, method, header, body):
    """Sends CRITICAL alerts to administrators via e-mail."""

    EMAIL_RECIPS = ["ops.team@ourcompany.com",]

    message = json.loads(body)

    send_mail(EMAIL_RECIPS, "CRITICAL ALERT", message)
    print ("Sent alert via e-mail! Alert Text: %s  " + \
           "Recipients: %s") % (str(message), str(EMAIL_RECIPS))

    channel.basic_ack(delivery_tag=method.delivery_tag)

def rate_limit_notify(channel, method, header, body):
    """Sends the message to the administrators via e-mail."""

    EMAIL_RECIPS = ["api.team@ourcompany.com",]

    message = json.loads(body)
```

通知处理程序

将消息从JSON解码

将邮件传输到SMTP服务器

确认消息

将消息从JSON解码

```
        #(f-asc_10) Transmit e-mail to SMTP server
        send_mail(EMAIL_RECIPS, "RATE LIMIT ALERT!", message)

        print ("Sent alert via e-mail! Alert Text: %s  " + \
               "Recipients: %s") % (str(message), str(EMAIL_RECIPS))

        channel.basic_ack(delivery_tag=method.delivery_tag)

if __name__ == "__main__":
        AMQP_SERVER = "localhost"
        AMQP_USER = "alert_user"
        AMQP_PASS = "alertme"
        AMQP_VHOST = "/"
        AMQP_EXCHANGE = "alerts"

        creds_broker = pika.PlainCredentials(AMQP_USER, AMQP_PASS)
        conn_params = pika.ConnectionParameters(AMQP_SERVER,
                                                virtual_host = AMQP_VHOST,
                                                credentials = creds_broker)
        conn_broker = pika.BlockingConnection(conn_params)

        channel = conn_broker.channel()

        channel.exchange_declare( exchange=AMQP_EXCHANGE,
                                  type="topic",
                                  auto_delete=False)

            channel.queue_declare(queue="critical", auto_delete=False)
            channel.queue_bind(queue="critical",
                               exchange="alerts",
                               routing_key="critical.*")
        channel.queue_declare(queue="rate_limit", auto_delete=False)
        channel.queue_bind(queue="rate_limit",
                               exchange="alerts",
                               routing_key="*.rate_limit")

        channel.basic_consume( critical_notify,
                               queue="critical",
                               no_ack=False,
                               consumer_tag="critical")

        channel.basic_consume( rate_limit_notify,
                               queue="rate_limit",
                               no_ack=False,
                               consumer_tag="rate_limit")

        print "Ready for alerts!"
        channel.start_consuming()
```

代理服务器设置

确认消息

建立到代理的连接服务器

声明交换器

构造队列，并绑定到topic交换器

设置告警处理程序

你现在拥有了一个优雅的消费者实现。它会简单地通过操作消息标记，针对不同的组将告警 AMQP 消息转换成 email 告警。添加额外的告警类型和传输方法也很简单。所有你要做的就是创建消费者回调函数以提供新的告警处理方法，并将其连接到新告警类型的绑定规则队列。如果没有消息给消费者处理的话，消费者就没什么用了。因此，让我们看看消费者是如何处理告警的。

在开始本节的时候，我们的目标是为现有的程序简化产生告警的过程。如果你看一下下列清单的话，就会发现：虽然消费者花费了 90 行代码来处理告警，但是告警的产生只用了 20 行代码。

清单 4.6 告警生成示例——alert_producer.py

```
import json, pika
from optparse import OptionParser

opt_parser = OptionParser()
opt_parser.add_option("-r",
                      "--routing-key",
                      dest="routing_key",
                      help="Routing key for message " + \
                      " (e.g. myalert.im)")
opt_parser.add_option("-m",
                      "--message",
                      dest="message",
                      help="Message text for alert.")

args = opt_parser.parse_args()[0]

creds_broker = pika.PlainCredentials("alert_user", "alertme")

conn_params = pika.ConnectionParameters("localhost",
                                        virtual_host = "/",
                                        credentials = creds_broker)
conn_broker = pika.BlockingConnection(conn_params)

channel = conn_broker.channel()

msg = json.dumps(args.message)
msg_props = pika.BasicProperties()
msg_props.content_type = "application/json"
msg_props.durable = False

channel.basic_publish(body=msg,
                      exchange="alerts",
                      properties=msg_props,
                      routing_key=args.routing_key)

print ("Sent message %s tagged with routing key '%s' to " + \
       "exchange '/'.") % (json.dumps(args.message),
                           args.routing_key)
```

读取命令行参数

建立到服务器的连接

将告警消息发送给服务器

可以从命令行运行该样例生产者，使用任何你想要的内容和路由标记来生成告警信息。该程序中的第一部分简单地从命令行获取了消息和路由键。你连接上了 RabbitMQ 服务器，和在告警消费者中使用的方式一样。当发布消息的时候，事情才变得有趣起来：

```
msg = json.dumps(args.message)
```

```
msg_props = pika.BasicProperties()
msg_props.content_type = "application/json"
msg_props.durable = False

channel.basic_publish(body=msg,
                      exchange="alerts",
                      properties=msg_props,
                      routing_key=args.routing_key)
```

总共五行代码就创建了告警信息并标记了合适的路由键（即 critical.mywebapp）。在你将告警消息的内容用 JSON 解码之后，就创建了一个名为 msg_props 的 BasicProperties 对象。该对象可以用来设置 AMQP 消息的可选内容类型头；同时，如果需要的话可以设置消息的持久化属性。最后，只需一行代码你就能将消息发往告警（alerts）交换器，并通过路由键来归类告警的类型。由于消息的路由键无法匹配任何绑定规则就会被丢弃，因此你可以为那些尚不支持的告警类型用路由键来进行标记。一旦你支持了这些告警类型之后，任何拥有这些路由键的告警信息将会路由到正确的消费者上。关于消费者最后有一点值得注意的是你传给 channel.basic_publish 的 block_on_flow_control 标记。这告诉 Pika，如果 RabbitMQ 流程控制机制需要停止发布的话，它需要暂缓从 basic_publish 返回。当 RabbitMQ 告知 Pika 可以继续的时候，它最终会返回，并允许继续发布。这使得生产者和 RabbitMQ 合作得很顺利，因此如果 Rabbit 变得不堪重负时，它会拧紧“龙头”，放慢生产者的脚步。如果是从其他无法忍受阻塞的程序来发布告警的话，需要确保 block_on_flow_control 设置成了 false。

在短短 100 行代码中，你的 Web 应用以灵活和可扩展的方式发送告警，并在之后异步地传输给接收者。当你需要传输信息并快速处理但又无须知道处理结果的时候，你也体会到了发后即忘的消息通信模式是多么的有益。举例来说，你可以轻而易举地扩展告警消费者，添加一个额外的处理器，使用 *.* 的模式将所有告警记录到数据库中。不过发后即忘的消息通信模式不仅仅只是告警和日志记录这些用途而已。让我们看一个需要在消息内容上执行 CPU 密集型处理的例子，以及 RabbitMQ 如何帮助你转到异步操作上去的。

4.2.2 并行处理

假设你开始运行自己的社交网络，并刚刚开发了一个闪亮的新功能：图片上传。人们想要分享他们与朋友以及家人一起的假日照片——也许你在哪里看到过呢。为

了增进用户之间的互动，当用户联系人中的某位上传了新的照片的时候，你想要通知他们的朋友。新功能发布后的一周，市场部的同事走到你的办公桌前要求你给用户一些积分，作为上传照片的奖励，以鼓励他们继续上传照片并在网站上保持活跃。你同意添加几行代码，并将一个奖励系统和上传照片处理程序挂钩。虽然看起来有点令人讨厌，但是它能如期工作使得老板对结果非常满意。

下个月带宽账单到了，那个运营部同事非常生气，因为带宽使用翻了整整一倍。提供给客户的外部 API 展现了全尺寸图片（本该提供指向较小的缩略图链接）。所以你最好让上传功能的代码生成缩略图。该怎么办？简单的方法就是往代码里再塞一点，直接就在上传控制器里执行缩略图生成，但是等等……如果每次照片上传你都得执行图片大小调整操作的话，这意味着前端 Web 服务器将会过载，所以你不能这样做。而且网站用户并不想要等待图片上传处理脚本，直到获得上传成功的确认信息。这意味着你需要一个更聪明的解决方法，它允许你并行地运行任务，同时需要在不同于服务请求的服务器上运行任务。

你发现调整图片大小、奖励用户积分以及通知用户朋友其实是三个独立的任务。这些任务之间没有相互依赖关系。它们不需要等待对方的结果才能继续运行。这意味着你可以重构代码，同时调整图片大小和其他事情。如果你完成了这样的设计，那你可以轻松应对新的需求了。需要记录每次图片上传？没问题，你只要添加一个新的工作者来处理日志就可以了。

这听起来不错，如梦境般；不过所有这些并行化的东西看起来难以实现。你需要做多少工作来完成消息多播？不会太多；只需输入 fanout 交换器即可。

我们之前讲过交换器类型。fanout 类型交换器会将消息的拷贝发送给绑定的队列，就是这么简单。利用它就可以实现照片上传模块功能了。每当用户上传照片时，你不是立即处理所有工作，而是会发布一条照片的元信息，然后让异步工作者来并行处理剩下的工作。RabbitMQ 会确保每个消费者都会得到一份消息的拷贝。处理消息的责任就落在了工作者的身上。

消息会包含以下关于照片的元信息：照片的图片 ID、用户 ID，以及用于定位图片在文件系统的路径。你将使用 JSON 作为数据交换的格式。将来如果你需要为不同的任务支持多种语言的话，JSON 会让工作变得简单。消息内容看起来像这样：

```
(
    'image_id': 123456,
```

```
    'user_id': 6543,
    'image_path': '/path/to/pic.jpg'
}
```

如图4.4所示，你声明了一个upload-pictures交换器，并绑定了三个队列：resize-picture、add-points和notify-friends。按照这个设计方案，你会发现增加一个新类型的任务，譬如日志记录，就只需要定义新的队列并将其绑定到upload-pictures交换器即可。作为开发者，你关注的焦点在于如何实现每个工作者以及消息发布的逻辑；RabbitMQ会负责剩下的部分。

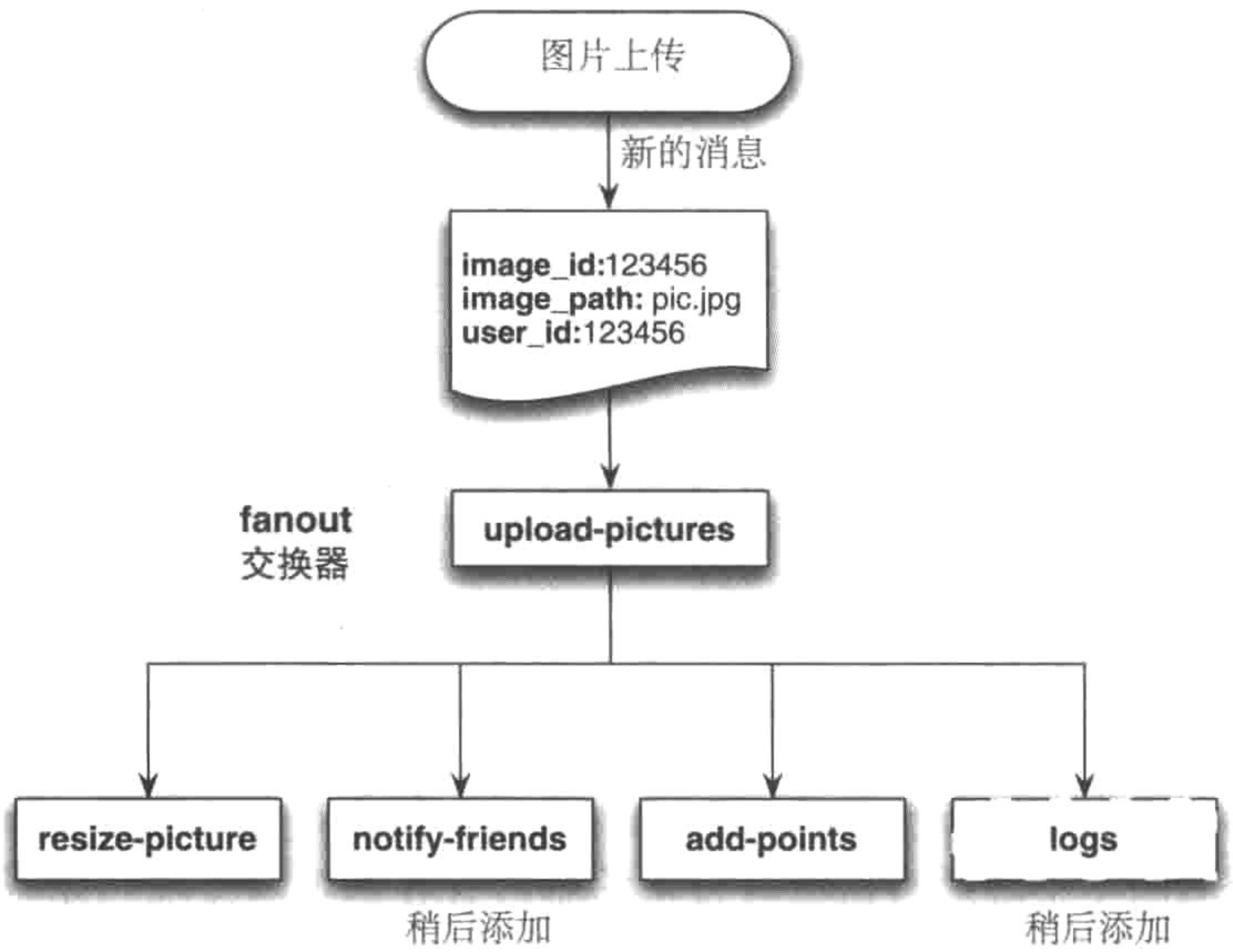

图4.4 上传图片

那么，让我们从添加发布者逻辑到照片上传模块开始吧，如下所示。这里你忽略了从POST请求接收图片，以及把它移动到文件系统的逻辑。

清单4.7 上传图片发布者

```
<?php

$channel->exchange_declare('upload-pictures',
  'fanout', false, true, false);

$metadata = json_encode(array(
      'image_id' => $image_id,
      'user_id' => $user_id,
      'image_path' => $image_path
      ));
```

❶ 声明交换器

❷ 将图片元数据编码为JSON

```
$msg = new AMQPMessage($metadata,
                       array('content_type' => 'application/json',
                             'delivery_mode' => 2));

$channel->basic_publish($msg, 'upload-pictures');
?>
```

❸ 实例化AMQP消息

❹ 发布消息

让我们看看你做了什么。这里没有关于获取 AMQP 信道的代码，这是因为我们在之前的例子中已经讲过了。在❶处，你声明了 upload-pictures 交换器，fanout 类型并且 durable 为 true。然后在❷处你创建了消息元数据，并编码为 JSON 格式。$image_id、_$user_id 和 $image_path 在上传过程中进行了初始化。在❸处你创建了新的消息实例，并将 deliver_mode 设置为 2 以使消息持久化。最终在❹处，你将消息发布到 upload-pictures 交换器。你不需要提供路由键，因为消息会被广播到所有绑定的队列。

下一步，让我们创建消费者，在每次上传后为用户添加积分。请参考 add-points-consumer.php 以获取完整代码。下列代码中省略了部分内容，因为我们之前已经讨论过，譬如包含 AMQP 库或者实例化连接和信道。

清单 4.8 添加积分消费者

```
<?php

$channel->exchange_declare('upload-pictures',
    'fanout', false, true, false);

$channel->queue_declare('add-points',
    false, true, false, false);

$channel->queue_bind('add-points', 'upload-pictures');

$consumer = function($msg){};

$channel->basic_consume($queue,
                        $consumer_tag,
                        false,
                false,
                false,
                false,
                $consumer);
?>
```

❶ 声明交换器

❷ 声明队列

❸ 绑定队列

❹ 部分代码省略了

❺ 开始消费消息

这段代码非常直白。在❶处你声明了 topic 类型交换器。然后在❷处你创建了 add-points 队列，RabbitMQ 会将消息投递到这里。在❸处你使用空的路由键将队列绑定到交换器上。在❹处，你暂时忽略了回调函数代码。在❺处你通过发送

basic_consume 命令来完成消费者的准备工作。同样地，你省略了等待循环以及信道和连接的清除代码。以下代码展示了回调函数。

清单 4.9 添加积分回调函数

```
<?php

function add_points_to_user($user_id){
    echo sprintf("Adding points to user: %s\n", $user_id);
}

$consumer = function($msg){

    if($msg->body == 'quit'){
        $msg->delivery_info['channel']->
            basic_cancel($msg->delivery_info['consumer_tag']);
    }

    $meta = json_decode($msg->body, true);

    add_points_to_user($meta['user_id']);

    $msg->delivery_info['channel']->
        basic_ack($msg->delivery_info['delivery_tag']);
};

?>
```

❶ 为用户添加积分函数

❷ 消费者回调函数

❸ 停止消费消息

❹ 解码JSON格式元数据

❺ 处理数据

确认消息

清单 4.9 显示的是真实处理消息的代码。在❶处你添加了哑函数，简单地将给用户的积分打印出来。在现实世界中，应用程序应该包含将用户积分添加到 Redis 数据库的逻辑。然后在❷处你定义了消费者回调函数。❸处的这段代码有点复杂，它是用来停止消费消息的钩子函数。如果消息体等同于 quit，那么就停止消费者。这段简单的代码确保了信道和连接以干净的方式关闭。之后在❹处你将消息体传入 json_decode 函数以获得元数据。你将方法的第二个参数设置为 true 以确保 PHP 会将 JSON 对象解码为关联数组。在❺处你调用了 add_points_to_user 函数，传入了从解码的消息中获得的 user_id。

让我们测试一下该实现代码。你将发布者代码复制过来并修改创建消息的逻辑，就得到了一份简单的测试脚本。在这种情况下，你从命令行获取三个参数：image ID、user ID 和 image path。你将它们进行编码之后，通过 RabbitMQ 发送给之前创建的消费者。我们不会解释下列清单，因为这和清单 4.7 看起来是一样的。

清单 4.10 上传图片测试

```
<?php
require_once('../lib/php-amqplib/amqp.inc');
require_once('../config/config.php');

$conn = new AMQPConnection(HOST, PORT, USER, PASS, VHOST);
$channel = $conn->channel();

$channel->exchange_declare('upload-pictures',
  'fanout', false, true, false);

$metadata = json_encode(array(
      'image_id' => $argv[1],
      'user_id' => $argv[2],
      'image_path' => $argv[3]
));

$msg = new AMQPMessage($metadata, array(
      'content_type' => 'application/json',
      'delivery_mode' => 2));

$channel->basic_publish($msg, 'upload-pictures');

$channel->close();
$conn->close();
?>
```

将这段代码保存在 fanout-publisher.php 文件中，并打开两个终端窗口。在第一个窗口，运行 add-points-consumer.php 脚本：

```
$ php add-points-consumer.php
```

在另一个窗口，执行发布者程序，传入一些随机参数以模拟请求：

```
$ php fanout-publisher.php 1 2 /path/to/pic.jpg
```

如果一切正常，你可以切回第一个终端并看到以下消息：

```
Adding points to user: 2
```

到目前为止，没什么印象深刻的。让我们添加另一个消费者，看看 fanout 交换器和实战并行处理。将下列清单中的代码放入 resize-picture-consumer.php 文件中。

清单 4.11 图片大小调整消费者

```
<?php

require_once('../lib/php-amqplib/amqp.inc');
require_once('../config/config.php');
```

```
$conn = new AMQPConnection(HOST, PORT, USER, PASS, VHOST);
$channel = $conn->channel();

$channel->exchange_declare('upload-pictures',
    'fanout', false, true, false);

$channel->queue_declare('resize-picture',
    false, true, false, false);

$channel->queue_bind('resize-picture', 'upload-pictures');

$consumer = function($msg){
    if($msg->body == 'quit'){
        $msg->delivery_info['channel']->
          basic_cancel($msg->delivery_info['consumer_tag']);
    }

    $meta = json_decode($msg->body, true);

    resize_picture($meta['image_id'], $meta['image_path']);

    $msg->delivery_info['channel']->
        basic_ack($msg->delivery_info['delivery_tag']);
};

function resize_picture($image_id, $image_path){
    echo sprintf("Resizing picture: %s %s\n",
        $image_id, $image_path);
}

$channel->basic_consume($queue,
                        $consumer_tag,
                        false,
                        false,
                        false,
                        false,
                        $consumer);

while(count($channel->callbacks)) {
    $channel->wait();
}

$channel->close();
$conn->close();
?>
```

❶ 声明图片调整大小队列

❷ 将队列绑定到交换器上

❸ 调整图片大小

❹ 图片调整函数

清单 4.11 的代码基本上和清单 4.8 很类似。有趣的地方在于❶和❷处，你创建了 resize-picture 队列，并绑定到了 upload-picture 交换器上。你会发现这里和前一个示例使用了同一个交换器。在 AMQP 中，一直都是把消息发布到一个交换器上，然后根据绑定，路由到一个或者多个队列中去（或者一个都去不了）。

代码接下来很直截了当；在消费者回调函数中，你将从元数据获得的 image_id 和 image_path 传入 resize_picture ❸ 的函数调用中。最终，resize_

picture 函数❹回显通知你它正在调整图片大小。和以往一样，在实际环境中，这里需要真实的图片大小调整代码。

现在，打开第三个终端窗口，并输入：

```
$ php resize-picture-consumer.php
```

然后回到之前的发布者脚本窗口，再次运行：

```
$ php fanout-publisher.php 1 2 /path/to/pic.jpg
```

如果一切正常的话，你应该看到每个消费者窗口都会显示以下信息：

```
Adding points to user: 2
```

和

```
Resizing picture: 1 /path/to/pic.jpg
```

从为用户添加积分这个消费者示例可以看出，如果将 RabbitMQ 集成到解决方案中去的话，扩展代码以应对新的需求是十分简单的。为了添加一个图片大小调整（image resize）消费者，你只需一个基于 image ID 和 path 的函数；可以从文件系统读取图片，对其调整大小（可能使用像 Imagemagick 这样的工具），然后基于 image ID 对数据库记录进行更新。对通知用户朋友来说也是一样的。将 user ID 作为参数，你就可以从数据库中获取用户的联系人，然后通过诸如电子邮件等形式发送通知给那些朋友。

从这个例子中你可以学到的是，RabbitMQ 消息通信的强大之处在于如何将交换器和队列进行组合。如果你需要某种方式过滤掉消息，那么可以使用之前章节提到的 topic 交换器。在应用程序中，是否存在一个动作触发其他动作，然后并行运行？如果有的话，使用 topic 交换器。如果你想“秘密监视”一段消息流，然后不留痕迹地离开，那么可以使用匿名队列并设置 autodeleted 为 true。一旦习惯思考消息通信模式，你将发现原来一些编程任务可以变得如此简单！

但是该设计方案的好处可不止于此。现在假设图片大小调整变得太慢了；你需要更多的计算能力，同时你又不想更改代码。小菜一碟。你可以启动更多的消费者进程，同时 RabbitMQ 会负责对消息进行相应的分发。即便消费者在不同的机器上，

这都没有问题。现在想象一下，当服务用户请求时，如果所有事情都按顺序发生的话，你该如何扩展原始代码呢？如你所见，RabbitMQ 并行处理是简单的。

4.3 别忘了：用RabbitMQ实现RPC并等待响应

有多种方式来实现远程过程调用（RPC）——譬如 UNIX RPC、REST API 和 SOAP。这些传统的 RPC 实现方法有共同之处：那就是客户端和服务器端紧密相连。客户端直接连接上服务器端，发送一个请求，然后就停下来等待服务器的应答。这种点对点性质的模式有很多好处，它使得在小范围内的拓扑变得简单。但这种简单的拓扑同时也限制了灵活性，并且当需要纵向扩展的时候，增加了复杂度。举例来说，当有众多服务器时，客户端如何发现在哪台服务器上可以找到其想要的服务呢？SOAP 和大多数企业 RPC 已经采取了复杂的补充协议和服务目录。这带来了额外复杂度和众多故障点。所有这些还都是以服务众多 RPC 服务器 API 而不会紧耦合客户端和服务端的名义发布的。此外，如果客户端连接的 RPC 服务器崩溃了，怎么办呢？这该由客户端来决定重连。并且，如果服务器端彻底停止运行的话，客户端必须重新发现能提供相同服务的新的服务器。当连上服务器端之后，客户端还不得不重试 API 调用。

如果取代复杂目录和多种协议而使用一种协议来进行 RPC 通信的话会如何呢？如果客户端可以发起 API 调用，而无须关心由哪台服务器来处理，同时也不必关心服务器崩溃的情况，那会如何呢？通过使用 MQ 服务器来实现 RPC 就可以做到这些。当使用 RabbitMQ 来实现 RPC 时，你只是简单地发布消息而已。RabbitMQ 会负责使用绑定来路由消息到达合适的队列。RPC 服务器会从这些队列上消费消息。RabbitMQ 替你完成了所有这些艰难的工作：将消息路由到合适的地方，通过多台 RPC 服务器对 RPC 消息进行负载均衡，甚至当处理消息的服务器崩溃时，将 RPC 消息重发到另一台。所有的这些都无须复杂的 WS-* 协议，也不需要客户端的路由功能。问题在于，如何将应答返回给客户端呢？毕竟，到目前为止你体验的 RabbitMQ 是发后即忘模型。

4.3.1 私有队列和发送确认

由于 AMQP 消息是单向的，RPC 服务器如何才能将结果返回给原始客户端呢？由于 RabbitMQ 处于中间环节，RPC 服务器甚至都不知道客户端调用者的身份（除非在消息体内放入应用特定的 ID）。谢天谢地，RabbitMQ 团队想出了一个优雅的解决方案：使用消息来发回应答。在每个 AMQP 消息头里有个字段叫作 `reply_to`。消息的生产者可以通过该字段来确定队列名称，并监听队列等待应答。然后接收消息的 RPC 服务器能够检查 `reply_to` 字段，并创建包含应答内容的新的消息，并以队列名称作为路由键。

你也许想："光是每次创建唯一队列名就得花很多工夫吧。我们怎样阻止其他客户端读到应答消息呢？"RabbitMQ 又一次拯救了你。你应该记得第 1 章的时候，如果你声明了没有名字的队列，RabbitMQ 会为你指定一个。这个名字恰好是唯一的队列名；同时在声明的时候指定 `exclusive` 参数，确保只有你可以读取队列上的消息。所有 RPC 客户端需要做的是声明临时的、排他的、匿名队列，并将该队列名称包含到 RPC 消息的 `reply_to` 头中，于是服务器端就知道应答消息该发往哪儿了。值得注意的是我们并没有提到将应答队列绑定到交换器上。这是因为当 RPC 服务器将应答消息发布到 RabbitMQ 而没有指定交换器时，RabbitMQ 就知道目的地是应答队列，路由键就是队列的名称。

话不多说。让我们看看在实际代码中如何使用 RabbitMQ 来实现 RPC。

4.3.2 使用 reply_to 来实现简单的 JSON RPC

首先你需要一台 RPC 服务器。在深入代码之前，先看一下 RPC 客户端和服务器端流程是很有帮助的，参见图 4.5。

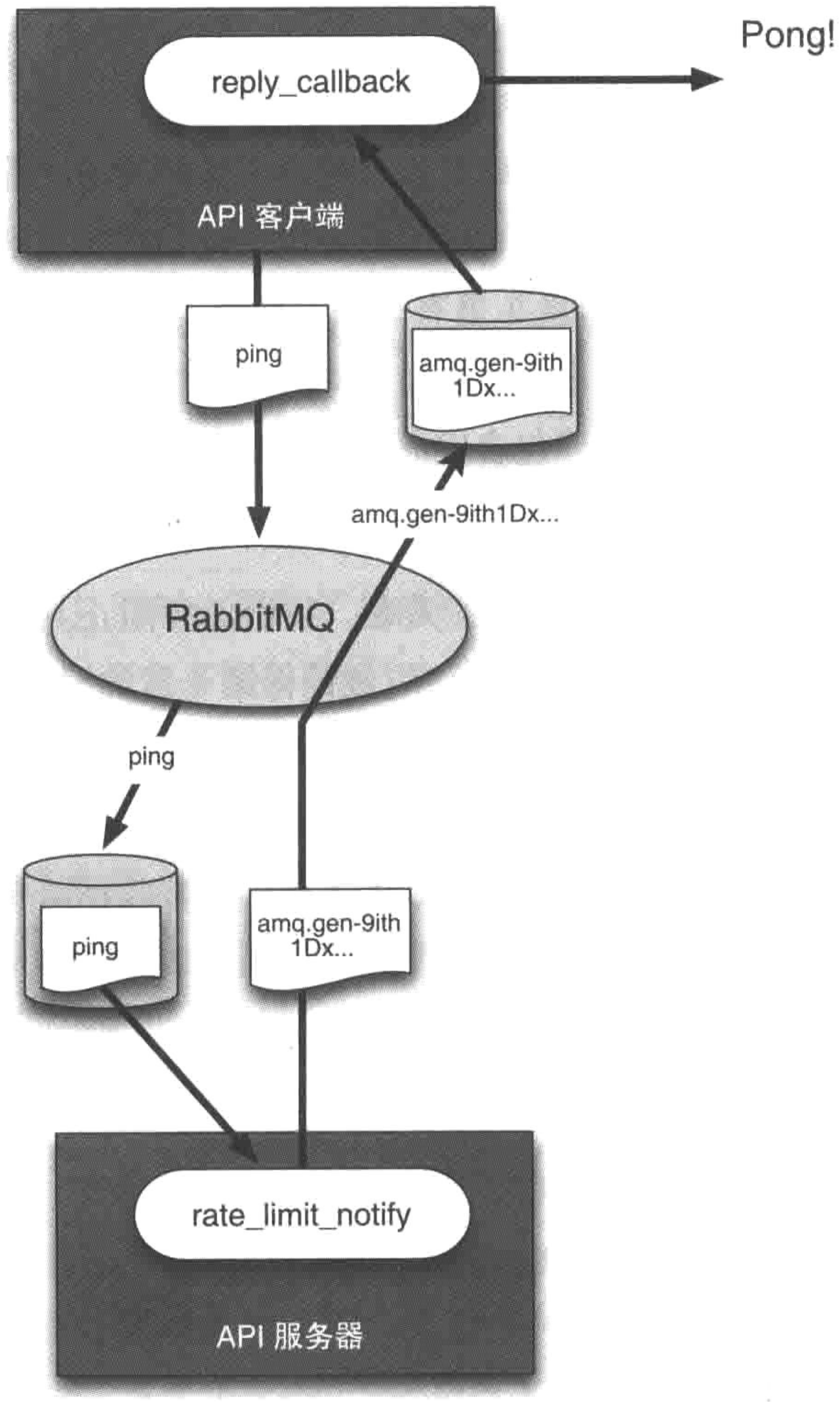

图 4.5 RPC 客户端和服务器流程

在下列清单中，你将构造一个简单的 API 服务器（实现了一个 ping 调用）。该方法的唯一功能是接收来自客户端的 ping 调用，发回一个 Pong! 应答，并携带了客户端初始调用时的时间戳。

清单 4.12 API 服务器——rpc_server.py

```
import pika, json

creds_broker = pika.PlainCredentials("rpc_user", "rpcme")
conn_params = pika.ConnectionParameters("localhost",
                                        virtual_host = "/",
                                        credentials = creds_broker)
conn_broker = pika.BlockingConnection(conn_params)
channel = conn_broker.channel()

channel.exchange_declare(exchange="rpc",
                         type="direct",
                         auto_delete=False)
channel.queue_declare(queue="ping", auto_delete=False)
channel.queue_bind(queue="ping",
                   exchange="rpc",
                   routing_key="ping")

def api_ping(channel, method, header, body):
    """'ping' API call."""
    channel.basic_ack(delivery_tag=method.delivery_tag)
    msg_dict = json.loads(body)
    print "Received API call...replying..."
    channel.basic_publish(body="Pong!" + str(msg_dict["time"]),
                          exchange="",
                          routing_key=header.reply_to)

channel.basic_consume(api_ping,
                      queue="ping",
                      consumer_tag="ping")

print "Waiting for RPC calls..."
channel.start_consuming()
```

建立到代理服务器的连接

声明交换器和ping调用队列

等待RPC调用和应答

我们已经讲过怎样配置和连接 RabbitMQ 服务器，所以让我们直接跳到最有趣的那部分：创建交换器和队列，用于接收 API 调用。

```
channel.exchange_declare(exchange="rpc",
                         type="direct",
                         auto_delete=False)
channel.queue_declare(queue="ping", auto_delete=False)
channel.queue_bind(queue="ping",
                   exchange="rpc",
                   routing_key="ping")
```

此处设置了典型的 direct 类型交换器并创建了队列和绑定。为了实现 API 服务器，你遵循着一种模式：将 RPC 函数调用的名称作为绑定模式（同时作为队列名称）。在该示例里，将 ping 队列绑定到 rpc 交换器并把 ping 作为绑定模式，ping API 函数调用就创建成功了。客户端需要做的是将 ping 作为路由键，同时将参数

放入消息体中。通过使用 topic 类型交换器，你可以使用更为复杂的 RPC 请求路由。下一步需要设置消费者订阅：

```
def api_ping(channel, method, header, body):
    """'ping' API call."""
    channel.basic_ack(delivery_tag=method.delivery_tag)
    msg_dict = json.loads(body)
    print "Received API call...replying..."
    channel.basic_publish(body="Pong!" + str(msg_dict["time"]),
                          exchange="",
                          routing_key=header.reply_to)

channel.basic_consume(api_ping,
                      queue="ping",
                      consumer_tag="ping")
```

每次 RabbitMQ 将消息通过 ping 队列送达时，api_ping 都会被调用。这里讲的和本书中到目前为止你做过的很相似。唯一的不同之处是在确认调用消息之后，你会发送一个 basic_publish 命令。等一下！你怎么能在用于消费的同一条信道上发布应答消息呢？我们不是说过这样不行吗？！事实上，在该示例中这是可能的，因为直到 api_ping 函数返回前，Pika 库不会再次开始消费。更值得注意的是 basic_publish 的配置。它使用消息头的 reply_to 作为应答消息的路由键。同时，不同于通过 RabbitMQ 发布的任何其他消息，这里没有交换器。这是关于通过 Rabbit 来实现 RPC 通信的唯一两处特别的地方：使用 reply_to 作为发布应答消息的目的地，同时发布的时候无须指定交换器。

那么 RPC 客户端是怎样的呢？它应该如何编写，如何设置应答队列呢？让我们看看下列清单。

清单 4.13 API 客户端——rpc_client.py

```
import time, json, pika

creds_broker = pika.PlainCredentials("rpc_user", "rpcme")
conn_params = pika.ConnectionParameters("localhost",
                                        virtual_host = "/",
                                        credentials = creds_broker)
conn_broker = pika.BlockingConnection(conn_params)
channel = conn_broker.channel()

msg = json.dumps({"client_name": "RPC Client 1.0",
                  "time" : time.time()})
result = channel.queue_declare(exclusive=True, auto_delete=True)
msg_props = pika.BasicProperties()
msg_props.reply_to=result.method.queue
```

建立到代理服务器的连接

发送RPC调用并等待应答

```
channel.basic_publish(body=msg,
                      exchange="rpc",
                      properties=msg_props,
                      routing_key="ping")

print "Sent 'ping' RPC call. Waiting for reply..."

def reply_callback(channel, method, header, body):
    """Receives RPC server replies."""
    print "RPC Reply --- " + body
    channel.stop_consuming()

channel.basic_consume(reply_callback,
                      queue=result.method.queue,
                      consumer_tag=result.method.queue)

channel.start_consuming()
```

客户端实现 RPC 的核心部分在此：

```
result = channel.queue_declare(exclusive=True, auto_delete=True)
msg_props = pika.BasicProperties()
msg_props.reply_to=result.queue
```

在这三行代码里，你创建了应答队列，并将消息的 reply_to 头设置为新队列的名称。当声明应答队列时，确保将 exclusive 和 auto_delete 都设置为 true。这样做可以确保没有人能够窃取你的消息(这是因为Rabbit创建的队列名是唯一的)，并且当接收完应答消息后断开和队列的连接时，Rabbit 会自动将队列删除。剩下的就是发布 API 调用消息，并将回调函数订阅到应答队列上：

```
channel.basic_publish(body=msg,
                      exchange="rpc",
                      properties=msg_props,
                      routing_key="ping")

print "Sent 'ping' RPC call. Waiting for reply..."

def reply_callback(channel, method, header, body):
    """Receives RPC server replies."""
    print "RPC Reply --- " + body
    channel.stop_consuming()

channel.basic_consume(reply_callback,
                      queue=result.method.queue,
                      consumer_tag=result.method.queue)
```

没什么神奇之处。一旦设置好应答队列，你就可以像其他队列那样从该队列进行消费了。只需确保从队列开始消费前，先发布 API 调用消息。否则的话，信道会置为消费模式，当你尝试发布消息时会得到一个错误。那么，当你运行 RPC 应用程

序时，客户端和服务器端会发生什么呢？

```
Client) Sent 'ping' RPC call. Waiting for reply...
Server) Received API call...replying...
Client) RPC Reply --- Pong! (Client Name: RPC Client 1.0)
          (RPC Call Issued Time: 1288111236.43)
```

你会发现这里服务器的应答就是针对该客户端的调用。因为服务器应答包含的时间戳就是客户端调用消息内的那个时间戳。在此，你可以轻易地通过创建队列和绑定的方式来扩展 API 以支持新的 API 方法。这样做最大的好处是任何一台服务器都无须对所有 API 调用做应答。举个例子，你可以简单地编写一个新的 RPC 服务器来进行图像处理，并运行在与 ping API 服务器不同的物理主机上。客户端不会知道有什么差异，你也可以根据需要随意扩展你的 API 服务。Rabbit 像施了魔法一样将所有这一切组织成一个 API 结构。没有特殊的协议和服务目录。

4.4 总结

在本章中，我们讨论了利用 RabbitMQ 编写应用程序的基础方法及其背后的消息通信模式。我们讨论了从发后即忘模式，譬如告警和图片处理，到使用真正的双向通信来实现 RPC API。在理解这些基础的消息通信架构后，你可以自由设计自己的复杂模式，将这些基础知识组合而成独一无二的解决方案来完成明确的目标。既然你开始用 RabbitMQ 来构建应用程序的核心的话，那么是时候让我们看看如何以弹性配置来运行 RabbitMQ 了，以确保当我们需要它的时候，它总是可用的。

5 集群并处理失败

本章要点

- RabbitMQ集群架构
- 在你的笔记本电脑上搭建集群
- 使用物理服务器创建集群
- 升级集群节点
- 与镜像队列一起工作

你刚刚依靠 RabbitMQ 消息队列的魔力，完成了了不起的新 Web 应用。用户接口展现了从后端 API 发送来的实时通知，并且 Rabbit 只发送每位 API 客户端感兴趣的通知。所有这一切看起来棒极了。同时 Rabbit 让你在老板面前看起来像一位编程大师。是时候该部署到产品环境了。你只需将 RabbitMQ 实例往生产环境服务器上一扔，一切就大功告成了，对吧？先别急。虽然对于客户端来说你的实时魔法看起来不错，但是当你的 RabbitMQ 服务器遇到内存崩溃了或者断电的情况，该怎么办呢？这款高性能的产品将成为公司的耻辱——也成为你的问题。现在正是时候讨论一下如何使得 RabbitMQ 能够弹性应对故障了，因而当墨菲定律肆虐你的应用程序时，你能确信作为应用程序“心脏”的 RabbitMQ 可以正常运行。

要让 RabbitMQ 变成高可用性有两种方法。其中一种是设置 Rabbit 集群，这样你可以从任何一台 Rabbit 故障中得以幸免，并且应用程序能够持续运作而不会停顿。另一种方法是扩大程序的规模以提升性能。单个 RabbitMQ 实例可能足够处理吉娃娃遛狗应用服务每天产生的消息吞吐量，但是当你需要处理每秒 1 000 000 条遛狗请求时，会发生什么呢？你需要 Rabbit 集群来保证应用程序正常运作。幸运的是，RabbitMQ 自带的内建集群可以解决以上两个问题。不管是因为服务器故障还是碰上了巨大的成功，你的应用程序总是能够连上 Rabbit。在本章中，我们会讲述 RabbitMQ 中令人着迷的集群。在本章结束的时候，你会理解集群背后的工作原理，并且明白如何在用于开发的笔记本电脑上搭建小规模的集群，也能在生产环境中搭建多台服务器集群。你会知道在新版本 Rabbit 发布后，如何升级集群。话不多说，让我们逐步深入，看看如何将几个 Rabbit 搭建成一个犀利的消息传递集群吧！

5.1 开足马力：RabbitMQ集群

RabbitMQ 最优秀的功能之一就是其内建集群。这使得它与几乎其他所有开源消息通信服务器有所区别；同时能够将集群在 5 分钟内搭建并运行起来这一事实，已经能将所有代理服务器“甩开好几条街”了。今天从第一台 Rabbit 启动开始，然后在运行时添加更多的 Rabbit 以增加高可用性或者提升性能，而完全不用停机。不过，RabbitMQ 集群并非“包治百病”。那么，RabbitMQ 集群可以带给你什么呢？

RabbitMQ 内建集群的设计用于完成两个目标：允许消费者和生产者在 Rabbit 节点崩溃的情况下继续运行，以及通过添加更多的节点来线性扩展消息通信吞吐量。RabbitMQ 通过利用 Erlang 提供的开放电信平台（Open Telecom Platform，OTP）分布式通信框架来巧妙地满足以上两个需求。你可以失去一个 RabbitMQ 节点，同时客户端能够重新连接到集群中的任何其他节点并继续生产或者消费消息，就像什么事都没发生过一样。同样地，如果 Rabbit 集群正疲于应对庞大消息通信量的话，那么添加更多的节点会线性地增加更多性能。RabbitMQ 集群不能保证消息的万无一失。

即便你将所有事情处理得很好（将消息、队列、交换器设置成可持久化，等等），当一个 Rabbit 集群节点崩溃时，该节点上队列的消息也会消失。这是因为 RabbitMQ 默认不会将队列的内容复制到整个集群上。如果不进行特别的配置，这些消息仅存在于队列所属的那个节点上。等一下，队列只存在于集群中的一个节点上？答案既是又非。为了能更好地理解，让我们看一下 RabbitMQ 集群架构。

5.2 集群架构

到目前为止，我们对于 RabbitMQ 的内部情况还不甚了解。确实，你知道队列和交换器的含义，以及如何将它们绑定到一起，也知道该使用哪些类型的交换器。但是这背后到底发生了什么呢？ RabbitMQ 是如何记录所有你使用的各种基础构件，同时它们又如何装配成一个消息通信服务器的呢？

RabbitMQ 会始终记录以下四种类型的内部元数据：

- 队列元数据——队列名称和它们的属性（是否可持久化，是否自动删除？）
- 交换器元数据——交换器名称、类型和属性（可持久化等）
- 绑定元数据—— 一张简单的表格展示了如何将消息路由到队列
- vhost 元数据——为 vhost 内的队列、交换器和绑定提供命名空间和安全属性

在单一节点内，RabbitMQ 会将所有这些信息存储在内存中，同时将那些标记为可持久化的队列和交换器（以及它们的绑定）存储到硬盘上。存储到硬盘上可以确保队列和交换器在重启 RabbitMQ 节点后重新创建。当你引入集群时，RabbitMQ 需要追踪新的元数据类型：集群节点位置，以及节点与已记录的其他类型元数据的关系。集群也提供了选择：将元数据存储到磁盘上（独立节点的默认设置），或者仅存储在 RAM 中。但是在我们深入到集群节点和它们如何存储元数据前，你应该首先理解在集群环境中队列和交换器的行为。

5.2.1 集群中的队列

在将两个节点组成集群的那一刻，事情发生了巨大的变化：不是每一个节点都有所有队列的完全拷贝。在单一节点设置中，所有关于队列的信息（元数据、状态和内容）都完全存储在该节点上（见图 5.1）。但是如果在集群中创建队列的话，集群只会在单个节点而不是在所有节点上创建完整的队列信息（元数据、状态、内容）。结果是只有队列的所有者节点知道有关队列的所有信息[1]。所有其他非所有者节点只知道队列的元数据和指向该队列存在的那个节点的指针。因此当集群节点崩溃时，该节点的队列和关联的绑定就都消失了。附加在那些队列上的消费者丢失了其订阅

1 在RabbitMQ 2.6.0及之后的版本中提供了镜像队列以允许队列内容免于集群节点故障。我们会在5.6节进行讲解。

信息，并且任何匹配该队列绑定信息的新消息也都丢失了。

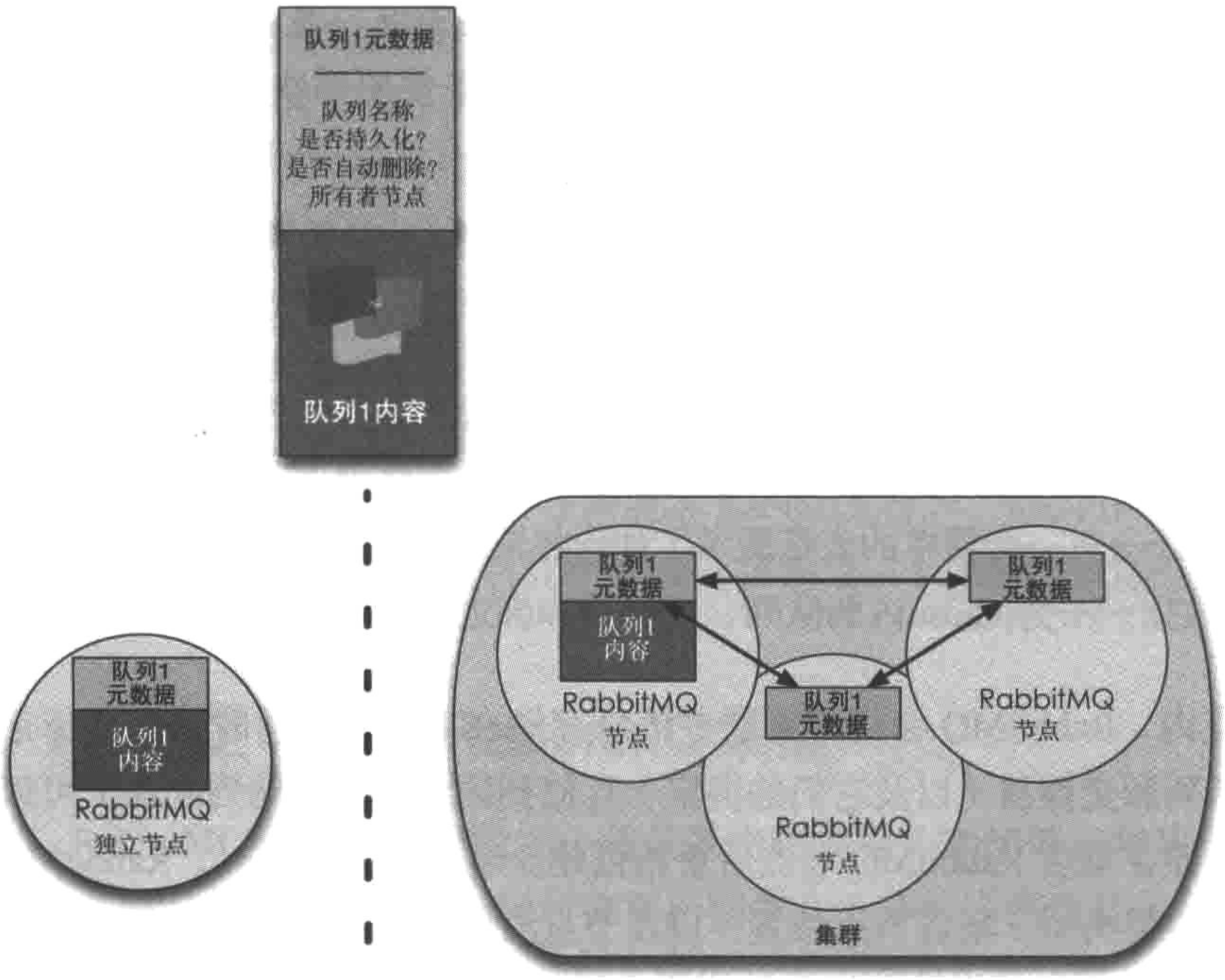

图 5.1 在独立节点和集群配置下的队列行为

别担心，你可以让消费者重连到集群并重新创建队列，对吧？这种做法仅当队列最开始没有被设置成可持久化时才是可行的。如果重新创建的队列被标记成持久化了，那么在其他节点上重新声明它们的话会得到一个难看的 404 NOT_FOUND 错误。这样确保了当失败节点恢复后加入集群，该节点上的队列消息不会丢失。想要该指定队列重回集群的唯一方法是恢复故障节点。但是如果消费者尝试重建的队列不是可持久化的，那么重新声明就会成功，你可以准备重新绑定它们并传输数据。挥之不去的问题是，为什么默认情况下 RabbitMQ 不将队列内容和状态复制到所有的节点上呢?

有两个原因：

（1）存储空间——如果每个集群节点都拥有所有队列的完整拷贝，那么添加新的节点不会给你带来更多存储空间。举个例子，如果一个节点可以存储 1GB 的消息，那么添加两个节点只会给你带来两个一模一样的 1GB 消息的拷贝。

（2）性能——消息的发布需要将消息复制到每一个集群节点。对于持久化消息来说，每一条消息都会触发磁盘活动。每次新增节点，网络和磁盘负载都会增加，最终只能保持集群性能的平稳（甚至更糟）。

通过设置集群中的唯一节点来负责任何特定队列，只有该负责节点才会因队列消息而遭受磁盘活动的影响。所有其他节点需要将接收到的该队列的消息传递给该队列的所有者节点。因此，往 Rabbit 集群添加更多的节点意味着你将拥有更多的节点来传播队列，这些新增节点为你带来了性能的提升。当负载增加时，RabbitMQ 集群是性能扩展的最佳方案。你可能好奇交换器是否也遵循同样的规则。并非如此，因为交换器本身是你凭空想象出来的。

5.2.2 分布交换器

到目前为止，我们一直把交换器描述得如同队列那样是一个活生生的实体。事实上，不同于队列那样拥有自己的进程，交换器说到底只是一个名称和一个队列绑定列表。当你将消息发布到交换器时，实际上是由你所连接到的信道将消息上的路由键同交换器的绑定列表进行比较，然后路由消息。正是信道（channel）按照绑定匹配的结果，将消息路由到队列。这是为什么呢？理解信道是真正的路由器这一点很重要，因为这解释了为什么交换器不会像集群中的队列那样受到相同的限制。

> **注意**：理解 RabbitMQ 背后的消息路由工作机制的方法是把每个队列想象成节点上运行的进程，每个进程拥有自己的进程 ID（PID）。交换器只不过是路由模式列表和匹配消息应发往的队列进程 ID 列表。当发布的消息匹配了交换器中的绑定规则时，实际上是由信道完成了匹配工作，并在匹配之后建立到队列 PID 的连接，然后将消息传送过去。队列的进程 ID 本质上是其在集群中的 Erlang 地址。

由于交换器只不过是一张查询表，而非实际上的消息路由器，因此将交换器在整个集群中进行复制会更加简单（见图 5.2）。举例来说，当创建一个新的交换器时，RabbitMQ 所要做的是将查询表添加到集群中的所有节点上。这时，每个节点上的每条信道都可以访问到新的交换器了。因此，相对于默认情况下队列的完整信息存在于集群中的单一节点来说，集群中的每个节点拥有每个交换器的所有信息。就可用性来讲，这非常棒，因为这意味着你不用担心在节点故障时重新声明交换器。只需让故障节点上的生产者重新连接到集群上，它们立即就能开始往交换器上发布消

息了。那么当消息已经发布到信道上，但在路由完成之前节点发生故障的话，这些消息会怎样呢？

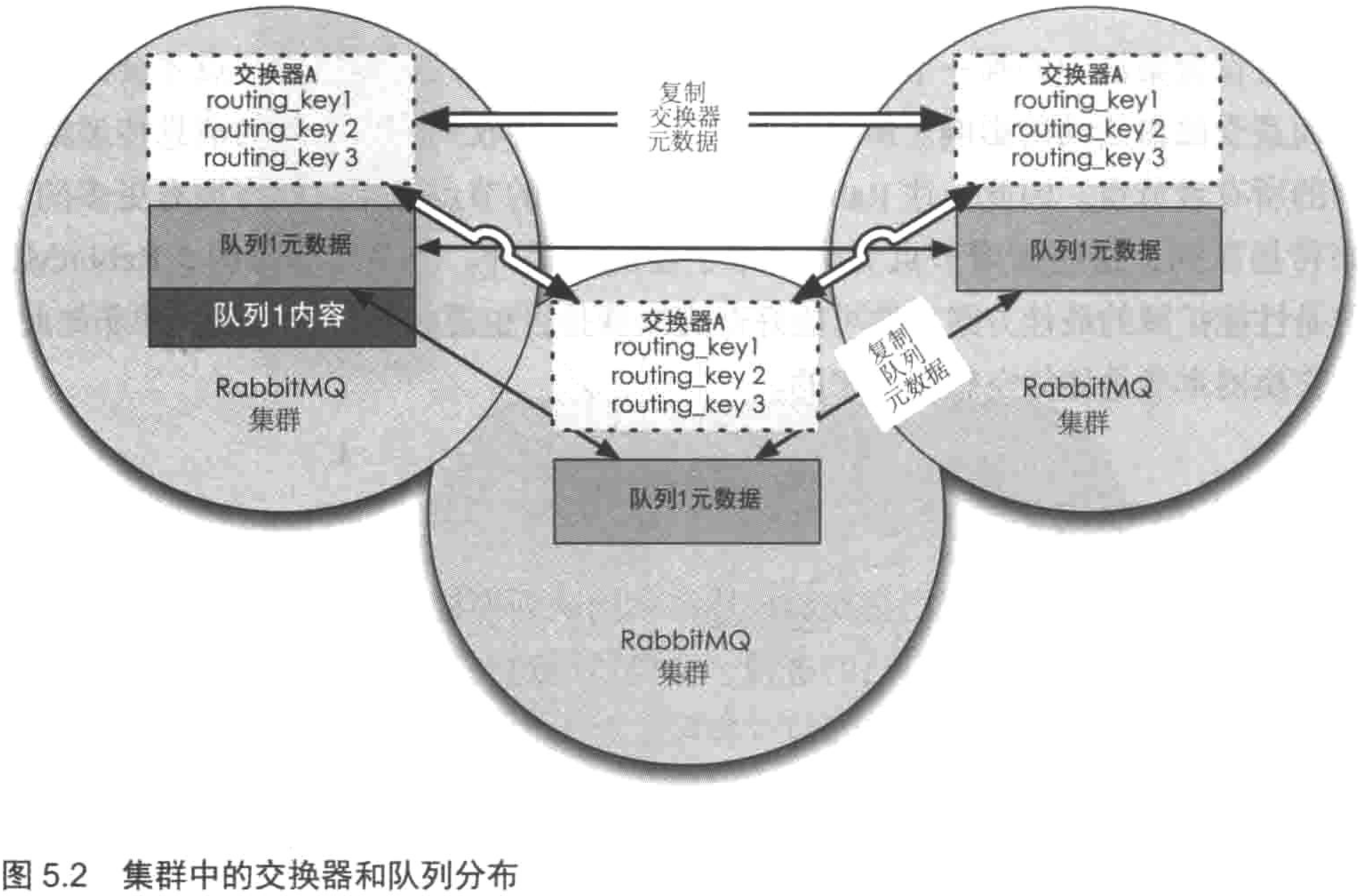

图 5.2 集群中的交换器和队列分布

AMQP 的 basic.publish 命令不会返回消息的状态。这意味着当信道节点崩溃时信道可能仍然在路由消息，而生产者已经继续创建下一条消息了。在这种情况下，你将承受丢失消息的风险。解决方法是使用 AMQP 事务，在消息路由到队列之前它会一直阻塞；或者使用发送方确认（publisher confirm）模式来记录连接中断时尚未被确认的消息。这两种解决方案也能帮助你在节点故障并且目的队列不复存在时检测到消息无法路由的情况。当与交换器在集群中完全复制的这一事实相结合时，事务模式和发送方确认模式都能确保应用程序一直发布而不丢失一条消息。

现在你明白了队列和交换器在集群中的表现，是时候看一下 RabbitMQ 是如何记录它们，以及节点如何为它们提供动力的了。

5.2.3 是内存节点还是磁盘节点

每个 RabbitMQ 节点，不管是单一节点系统或者是庞大集群的一部分，要么是内存节点（RAM node），要么是磁盘节点（disk node）。内存节点将所有的队列、交

换器、绑定、用户、权限和 vhost 的元数据定义都仅存储在内存中。而磁盘节点则将元数据存储在磁盘中（见图 5.3）。单节点系统只允许磁盘类型的节点；否则，每次你重启 RabbitMQ 之后，所有关于系统的配置信息都会丢失。不过在集群中，你可以选择配置部分节点为内存节点。为什么会想要选择将元数据仅存储在内存中？因为它使得像队列和交换器声明之类的操作更加快速。

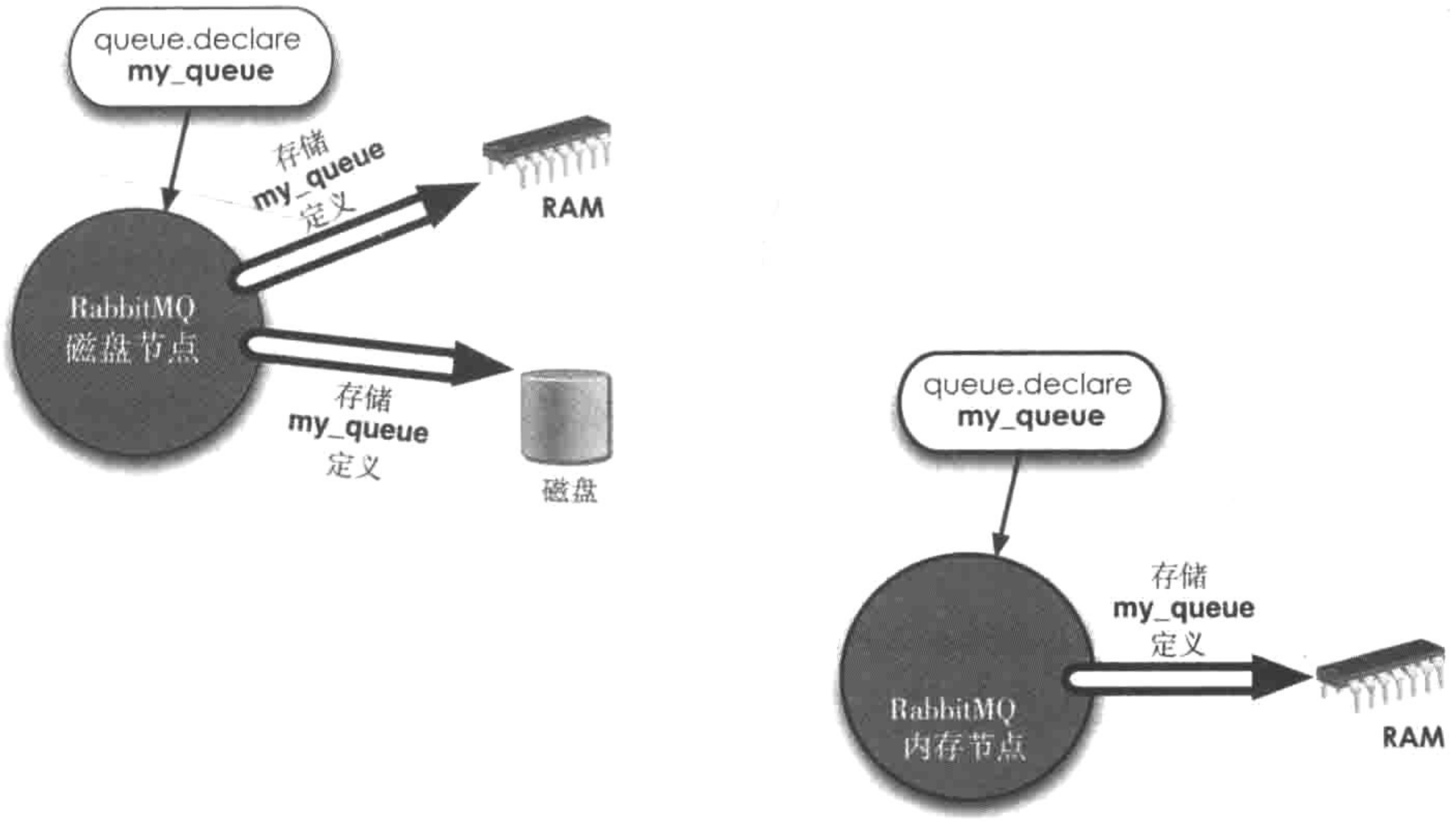

图 5.3 元数据写入内存和磁盘节点

当在集群中声明队列、交换器或者绑定的时候，这些操作会直到所有集群节点都成功提交元数据变更后才返回。对于内存节点来说，这意味着将变更写入内存；而对于磁盘节点来说，这意味着昂贵的磁盘写入操作，直到完成之后，节点才能说："完事儿了！"假设你有五个节点的集群，并且所有节点都是磁盘节点，则你必须得等待所有这五个节点将元数据写入磁盘后，队列声明操作才能返回。对于队列长存的服务器来说这没什么大不了的。但是如果你正处理繁重的 RPC 呢？如果每个 RPC 客户端正创建和销毁每秒数以百计的应答队列，你就会明白磁盘节点会降低性能。内存节点可以提供的出色性能，和磁盘节点能够保障集群配置信息幸免于重启，你如何平衡这两者呢？

RabbitMQ 只要求在集群中至少有一个磁盘节点。所有其他节点可以是内存节点。记住，当节点加入或者离开集群时，它们必须要将该变更通知到至少一个磁盘节点。如果只有一个磁盘节点，而且不凑巧的是它刚好又崩溃了，那么集群可以继续路由

消息，但是不能做以下的操作了：

- 创建队列
- 创建交换器
- 创建绑定
- 添加用户
- 更改权限
- 添加或删除集群节点

换句话说，如果集群中唯一的磁盘节点崩溃的话，集群仍然可以保持运行，但是直到将该节点恢复到集群前，你无法更改任何东西。解决方案是在集群中设置两个磁盘节点，因此它们中至少有一个是可用的，能在任何时候保存元数据变更。只有一个需要所有的磁盘节点必须在线的操作是添加或者删除集群节点。当内存节点重启后，它们会连接到预先配置的磁盘节点，下载当前集群元数据拷贝。如果你只将两个磁盘节点中的一个告诉了该内存节点，而不凑巧的是当内存节点供电中断时该磁盘节点也发生故障的话，那么内存节点在重启之后就无法找到集群了。所以当添加内存节点时，确保告知其所有的磁盘节点（内存节点唯一存储到磁盘的元数据信息是集群中磁盘节点的地址）。只要内存节点可以找到至少一个磁盘节点，那么它就能在重启后重新加入集群。

话不多说。让我们开始建立集群吧！

5.3 在你的笔记本电脑上设置集群

理解 RabbitMQ 如何处理集群的内部工作机制是比较困难的；不过设置集群倒是非常简单！简单到可以在你的开发系统中设置完整功能的集群。这非常的棒，因为这意味着在你编写代码时，你可以在生产环境中，测试失败场景，并观察它们是如何处理的。在第 1 章，你应该将 RabbitMQ 服务器安装到你的开发机器上了。在开始配置集群前，需要首先确保现存的 Rabbit 没有运行。如果需要关闭节点的话，则移动到 RabbitMQ 安装目录，然后运行 `sbin/rabbitmqctl stop`[2]。你应该会看到如下信息，提示你节点已经关闭了。

2 通过使用RabbitMQ中央配置文件替代`rabbitmqctl`来配置集群成员也是可以的。由于这个方法容易出现错误并且难以调试，因此我们不会对它进行介绍。

```
Stopping and halting node rabbit@Phantome ... ...done.
```

现在节点已经关闭了，你可以开始配置集群了。通常来讲，你使用 rabbitmq-server 命令启动节点之后就大功告成了。但是如果不用额外参数的话，该命令会默认使用节点名称 rabbit 和监听端口号 5672。如果你尝试在同台机器用这个方法启动三个节点的话，那么第二个和第三个节点会因为节点名称和端口冲突导致启动失败。在每次调用 rabbitmq-server 命令前，我们通过设置 RABBITMQ_NODENAME 和 RABBITMQ_NODE_PORT 环境变量来明确指定唯一的节点名称和端口号。在我们的例子中，端口号从 5672 开始，接着每个新启动的节点都增加 1。同样地，第一个节点命名为 rabbit，之后第二个和第三个节点则分别命名为 rabbit_1 和 rabbit_2。你可以使用不同的节点名称和端口号，只要对每个节点来说是唯一的即可。让我们在开发系统上设置并启动三个节点，如下列清单所示（首先，确保移除 RabbitMQ【2.7.0 之前的版本】所有插件，禁用 RabbitMQ【2.7.0 和 2.7.0 之后的版本】所有插件）。

注意：我们尚未讨论 RabbitMQ 插件，不过有可能你已经启用了一部分插件了。如果确实如此的话，你需要在启动集群节点前将插件禁用。这是因为像 RabbitMQ Management 插件这样的插件会监听专门的端口来提供服务（例如 Management 插件的 Web UI）。我们还没有讲到如何设置插件监听不同的端口，所以当第二个和之后的节点启动了它们的插件，就会和第一个启动节点的插件相冲突，然后节点就都崩溃了。

清单 5.1 在开发系统上启动三个节点的集群

```
$ RABBITMQ_NODE_PORT=5672 RABBITMQ_NODENAME=rabbit \
  ./sbin/rabbitmq-server -detached
Activating RabbitMQ plugins ...
0 plugins activated:

$ RABBITMQ_NODE_PORT=5673 RABBITMQ_NODENAME=rabbit_1 \
  ./sbin/rabbitmq-server -detached
Activating RabbitMQ plugins ...
0 plugins activated:

$ RABBITMQ_NODE_PORT=5674 RABBITMQ_NODENAME=rabbit_2 \
  ./sbin/rabbitmq-server -detached
Activating RabbitMQ plugins ...
0 plugins activated:
```

现在在开发系统上运行着三个 Rabbit 节点，分别是 rabbit、rabbit_1 和

rabbit_2（每个节点都会有系统的主机名，添加在@后）。但是每个节点仍然是独立节点，拥有自己的元数据，并且不知道其他节点的存在。集群中的第一个节点将初始元数据带入集群中，并且无须被告知加入。而第二个和之后的节点将加入它并且获取它的元数据。要加入第二个和第三个节点，首先需要停止 Erlang 节点上运行的 RabbitMQ 应用程序，并重设（清空）它们的元数据。这样它们才可以被加入并且获取集群的元数据。rabbitmqctl 工具会帮你和每个节点通信并完成以上的任务。让我们从停止第二个节点上的 RabbitMQ 应用程序开始：

```
$ ./sbin/rabbitmqctl -n rabbit_1@Phantome stop_app
Stopping node rabbit_1@Phantome ...
...done.
```

下一步，你需要重设第二个节点的元数据和状态为清空的状态：

```
$ ./sbin/rabbitmqctl -n rabbit_1@Phantome reset
Resetting node rabbit_1@Phantome ...
...done.
```

现在拥有了一个停止运行的（并且清空了的）Rabbit 应用，你已经准备好将它加入到第一个集群节点：

```
$ ./sbin/rabbitmqctl -n rabbit_1@Phantome cluster rabbit@Phantome \
                                                  rabbit_1@Phantome
Clustering node rabbit_1@Phantome with [rabbit@Phantome,
                                        rabbit_1@Phantome] ...
...done.
```

最后你可以重新启动第二个节点的应用程序，这样它就开始成为集群中运作的一员了：

```
$ ./sbin/rabbitmqctl -n rabbit_1@Phantome start_app
Starting node rabbit_1@Phantome ...
...
broker running
...done.
```

你也许注意到了，当你发送 cluster 命令式到第二个节点时，你获得了 Clustering node rabbit_1@Phantome with [rabbit@Phantome, rabbit_1@Phantome] 应答。这很奇怪；除了第一个节点之外，你还将第二个节点自己也集群进来？这样做是对的。原因在于你想让 rabbit_1 也成为磁盘节点。当你将新节点加入到集群时，你必须列出在集群中的所有磁盘节点，并作为集群命令的参数。这就是内存节点在重启之后还能知道如何获得初始元数据和状态的原因。

如果磁盘节点之一就是要新增节点的话，rabbitmqctl 可以非常机智地认识到你想让该新增节点也成为磁盘节点。另一个传入 rabbitmqctl 命令的关键参数是 -n rabbit_1@Phantome。这会告诉 rabbitmqctl 你想要在指定节点而非默认节点（rabbit@）上执行命令。你可以使用 -n 参数指定任何 RabbitMQ 节点，不管是在你的开发系统上或者是网络上任何可以连接的其他系统上。

> **注意**：记住 Erlang 节点间通过认证 Erlang cookie 的方式来允许互相通信。因为 rabbitmqctl 使用 Erlang OTP 通信机制来和 Rabbit 节点通信，运行 rabbitmqctl 的机器和所要连接的 Rabbit 节点必须使用相同的 Erlang cookie。否则你会得到一个错误。

到目前为止你的开发系统上已经有了两个节点的 Rabbit 集群，第三个独立 Rabbit 节点正等待加入集群。别把这第三个节点留在冷风中痴痴地等！如下列清单所示，加入第三个节点和加入第二个节点的命令几乎是一样的。

清单 5.2 将第三个节点加入集群

```
$ ./sbin/rabbitmqctl -n rabbit_2@Phantome stop_app
Stopping node rabbit_2@Phantome ...
...done.
$ ./sbin/rabbitmqctl -n rabbit_2@Phantome reset
Resetting node rabbit_2@Phantome ...
...done.
$ ./sbin/rabbitmqctl -n rabbit_2@Phantome cluster rabbit@Phantome \
                                                  rabbit_1@Phantome
Clustering node rabbit_2@Phantome with [rabbit@Phantome,
                                        rabbit_1@Phantome] ...
...done.
$ ./sbin/rabbitmqctl -n rabbit_2@Phantome start_app
Starting node rabbit_2@Phantome ...
...
broker running
...done.
```

当你使用相同的命令将第三个节点加入时，唯一不同的是 -n 参数指定了第三个节点。你也许注意到了你指定了 rabbit 和 rabbit_1 作为集群命令的参数，但却没有 rabbit_2。因此，rabbit_2 会知道集群中的两个磁盘节点，但其本身不会成为磁盘节点。取而代之的是，通过不指定其作为参数，rabbit_2 会成为内存节点。

在所有节点运行并成功集群之后，让我们看看你的作品，并用 rabbitmqctl

来查看集群。

```
$ ./sbin/rabbitmqctl cluster_status
        Cluster status of node rabbit@Phantome ...
        [{nodes,[{disc,[rabbit_1@Phantome,rabbit@Phantome]},
                 {ram,[rabbit_2@Phantome]}]},
         {running_nodes,[rabbit_1@Phantome,rabbit@Phantome]}]
        ...done.
```

关键是 nodes 部分：

```
{nodes,[{disc,[rabbit_1@Phantome,rabbit@Phantome]},
          {ram,[rabbit_2@Phantome]}]},
```

rabbitmqctl 告诉你有三个节点加入到了集群。

- 两个（“disc”）节点：rabbit 和 rabbit_1
- 一个内存节点：rabbit_2

running_nodes 部分告诉你集群中的哪些节点正在运行。现在你可以连接到这三个 running_nodes 中的任何一个，并且开始创建队列，发布消息或者执行其他任何 AMQP 任务。但在开始使用集群学习如何编写可以重连并处理节点故障的程序之前，你应该将新学到的集群搭建技能用于在多台电脑上创建集群。

5.4 将节点分布到更多的机器上

将 RabbitMQ 集群运行在多台物理机器上并不比在开发系统上构建集群难多少。首先你需要知道的是 RabbitMQ 集群对延迟非常敏感，应当只在本地局域网内使用。使用集群来提供地理可用性或者通过 WAN 来路由消息，都会导致超时和怪异的集群行为，因此这是不明智的。记住这些，我们将在本地局域网创建分布式集群，像这样：

- 三个节点分别在三台独立的物理机器上（Amazon EC2 微型实例 / 服务器）
- 每个节点在 Ubuntu 10.04 LTS 64-bit 上运行 RabbitMQ 2.7.0
- Erlang R13B04

---

3 作者笔误：根据上下文，应为{running_nodes,[rabbit_2@Phantome,rabbit_1@Phantome,rabbit@Phantome]}]——译者注

首先你需要先在机器上安装操作系统和 RabbitMQ 2.7.0（在每台系统上使用第 1 章中的指令）。虽然我们使用的是 Ubuntu，但是在任何基于 UNIX 的操作系统上都可以工作。同时，为了使得设置这三个系统尽量简单，我们使用了 Amazon Web Services EC2 服务器（http://aws.amazon.com/ec2）。我们的 EC2 服务器是拥有 613 MB 内存和 8 GB 存储空间的微型实例。因此你可以专注于真正的集群工作上而非服务器设置上，我们制作了装有 Ubuntu 和 RabbitMQ 的 Amazon Machine Image（AMI），以便随时可以用来集群。要使用它的话，启动三台 EC2 US West - N. California 区域的新服务器，并在选择服务器镜像的时候搜索 AMI ID `ami-69ebb42c`。看吧！你应该有了三台闪亮的新的微型 EC2 服务器，运行着 Ubuntu 10.04 LTS 和一份纯净的 RabbitMQ 2.7.0 拷贝[4]。

我们将三台服务器分别设置为 `ip-10-170-29-145`、`ip-10-170-30-18` 和 `ip-10-170-29-88`（见图 5.4）[5]。当从 AMI 创建服务器时，Amazon Web Services 会自动分配主机名。你的主机名会有所不同，因此使用分配的主机名来代替即可。

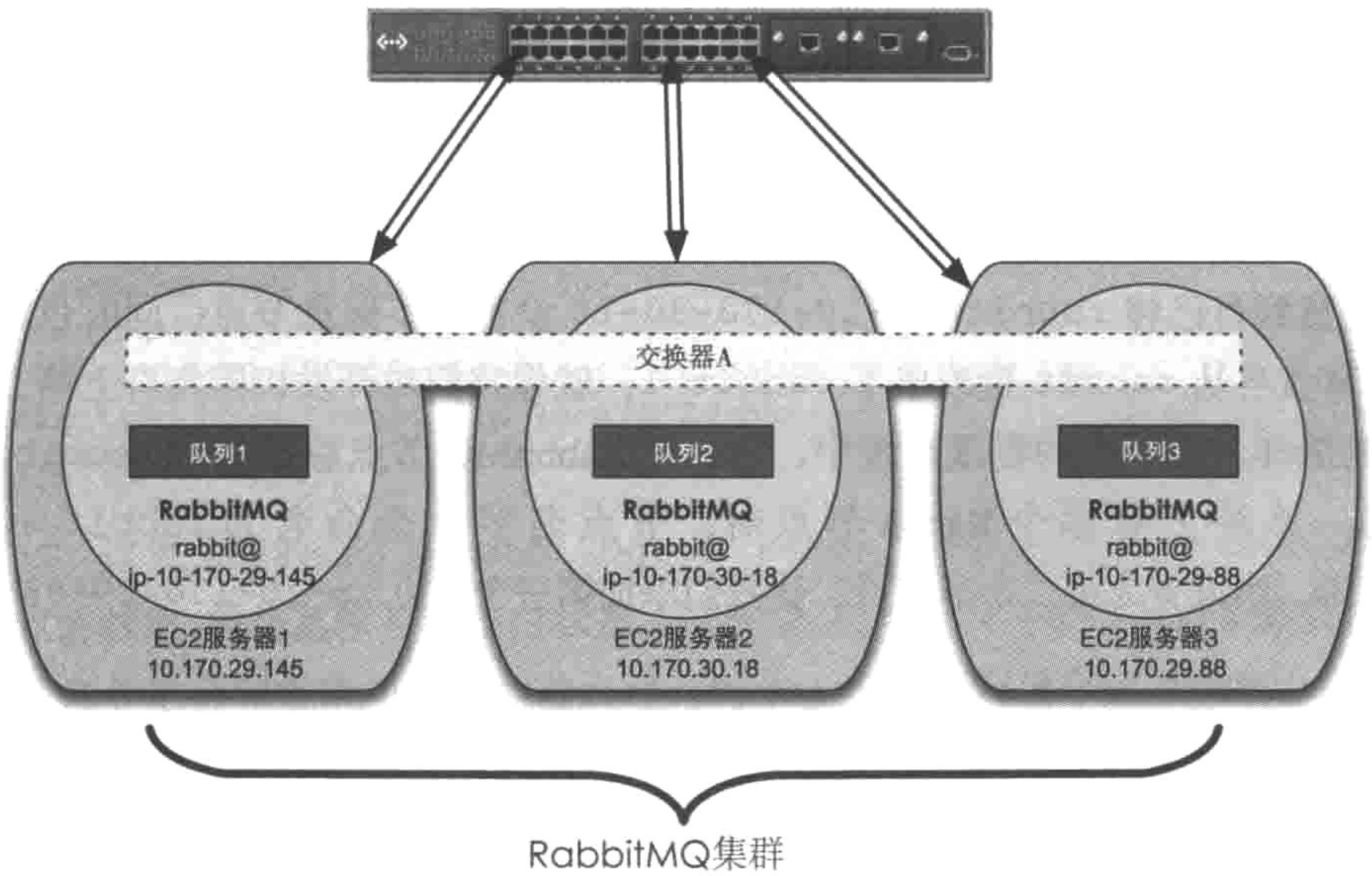

图 5.4 在 EC2 服务器上构建 Rabbit 集群

4 如果你使用我们的AMI来搭建EC2的话，在使用SSH连接到服务器时确保使用`ubuntu`作为用户名。

5 作者笔误：英文原版书中图5.4上的主机名和IP地址无法与文中内容对应起来。根据文中作者提到的机器名，中文翻译版中已对此进行了修改。——译者注

现在，你需要做的是将 ip-10-170-29-145 的 Erlang cookie 复制到其他几个节点，让它们可以互相通信。如果节点的 Erlang cookie 字串不同，那么当 Erlang 节点尝试互相认证时会导致集群失败。如果你使用的是我们的 AMI，那么你可以在 /var/lib/rabbitmq/.erlang.cookie 找到 Erlang cookie。复制 cookie 内的字串，并粘贴到其他两个节点的 /var/lib/rabbitmq/.erlang.cookie 上。然后通过运行 sudo /etc/init.d/rabbitmq-server restart 命令重启另外两个节点的 RabbitMQ 进程。这其实是构建分布式集群中最困难的部分了。现在可以准备加入集群了。首先将 ip-10-170-30-18 加入集群。

```
ubuntu@ip-10-170-30-18:~# sudo rabbitmqctl stop_app
Stopping node 'rabbit@ip-10-170-30-18' ...
...done.
ubuntu@ip-10-170-30-18:~# sudo rabbitmqctl reset
Resetting node 'rabbit@ip-10-170-30-18' ...
...done.
ubuntu@ip-10-170-30-18:~# sudo rabbitmqctl cluster \
                                    rabbit@ip-10-170-29-145 \
                                    rabbit@ip-10-170-30-18
Clustering node 'rabbit@ip-10-170-30-18' with
                                   ['rabbit@ip-10-170-29-145',
                                    'rabbit@ip-10-170-30-18'] ...
...done.
ubuntu@ip-10-170-30-18:~# sudo rabbitmqctl start_app
Starting node 'rabbit@ip-10-170-30-18' ...
...done.
```

你会注意到自己将 rabbit@ip-10-170-30-18 设置成了磁盘节点，同时也将 RabbitMQ 节点名从 rabbit 更改成了 rabbit_1，就像我们对开发机器集群上第二个节点做的那样。在隔离的物理系统中，第一个 RabbitMQ 节点总是叫作 rabbit。只有当同一系统上有多个 Rabbit 节点时，节点名字才会命名为 rabbit_1、rabbit_2 等。当设置完第二个节点后，让我们添加第三个节点 ip-10-170-29-88：

```
ubuntu@ip-10-170-29-88:~$ sudo rabbitmqctl stop_app
Stopping node 'rabbit@ip-10-170-29-88' ...
...done.
ubuntu@ip-10-170-29-88:~$ sudo rabbitmqctl reset
Resetting node 'rabbit@ip-10-170-29-88' ...
...done.
ubuntu@ip-10-170-29-88:~$ sudo rabbitmqctl cluster \
                                    rabbit@ip-10-170-29-145 \
                                    rabbit@ip-10-170-30-18
Clustering node 'rabbit@ip-10-170-29-88' with
                                   ['rabbit@ip-10-170-29-145',
                                    'rabbit@ip-10-170-30-18'] ...
```

```
...done.
ubuntu@ip-10-170-29-88:~$ sudo rabbitmqctl start_app
Starting node 'rabbit@ip-10-170-29-88' ...
...done.
```

如果在任何一个节点上运行 sudo rabbitmqctl cluster_status，你会看到现在自己有了三个节点的集群了：

```
Cluster status of node rabbit@ip-10-170-29-88 ...
 [{nodes,[{disc,['rabbit@ip-10-170-30-18','rabbit@ip-10-170-29-145']},
          {ram,['rabbit@ip-10-170-29-88']}]},
  {running_nodes,['rabbit@ip-10-170-29-145','rabbit@ip-10-170-30-18',
                  'rabbit@ip-10-170-29-88']}]
...done.
```

到目前为止，你已经搭建了两种不同的 RabbitMQ 集群了：一种分布在不同的服务器上，另一种则在单台机器上。不过有一件事我们还没有介绍，那就是如何将节点从集群中移除。如果你想要让集群规模更小，或者用更好的硬件来替换一个节点的时候，该怎么做呢？针对这两种情况，你需要做的是让节点离开集群。和加入集群的操作类似，只是不需要 rabbitmqctl cluster 这一步了。让我们将 ip-10-170-29-88 从集群中移除，并将其转回独立节点：

```
ubuntu@ip-10-170-29-88:~$ sudo rabbitmqctl stop_app
Stopping node 'rabbit@ip-10-170-29-88' ...
...done.
ubuntu@ip-10-170-29-88:~$ sudo rabbitmqctl reset
Resetting node 'rabbit@ip-10-170-29-88' ...
...done.
ubuntu@ip-10-170-29-88:~$ sudo rabbitmqctl start_app
Starting node 'rabbit@ip-10-170-29-88' ...
...done.
```

这里关键的命令是 rabbitmqctl reset。我们之前说过 reset 命令将清空节点的状态，并将其恢复到空白状态。这没错，只不过当重设的节点是集群的一部分时，该命令也会和集群中的磁盘节点进行通信，告诉它们该节点正在离开集群。这很重要；不然，集群会认为该节点出了故障，并期望其最终能恢复回来。当离开的节点是磁盘节点时，正式离开集群就显得格外重要。回忆之前提到的，对于每次元数据变更来说，磁盘节点是必需的；但是对于节点加入或是离开集群来说，所有磁盘节点都是必需的。所以如果是非正式移除磁盘节点的话，集群会认为该节点发生故障，并等待其恢复后才允许新节点的加入。因此，简单地把磁盘节点从集群中猛拉出来而非正式移除的话，会导致集群永久性无法进行变更。所以当从集群中移除节点时，

请始终小心重设节点状态。

如果从移除的节点上检查集群的状态，你会发现它现在是独立节点。

```
Cluster status of node rabbit@ip-10-170-29-88 ...
 [{nodes,[{disc,['rabbit@ip-10-170-29-88']}]},
  {running_nodes,['rabbit@ip-10-170-29-88']}]
...done.
```

同样地，如果你从集群中其他剩余节点上来检查集群状态的话，你会看到它们已不再将 ip-10-170-29-88 视为集群的一部分了：

```
Cluster status of node rabbit@ip-10-170-30-18 ...
 [{nodes,[{disc,['rabbit@ip-10-170-30-18','rabbit@ip-10-170-29-145']}]},
  {running_nodes,['rabbit@ip-10-170-29-145','rabbit@ip-10-170-30-18']}]
...done.
```

在掌握了分布式集群以及如何正式地移除节点后，让我们讨论如何将集群升级到 RabbitMQ 新版本。

5.5 升级集群节点

通常来讲，在独立系统中升级到新版本 RabbitMQ 很容易。你只需解压新版本，然后运行即可[6]。旧的数据会被保留，而你将运行最新、最棒的 RabbitMQ 版本。但是升级集群并非如此简单。集群的升级是半自动化的。如果简单地将新版本 RabbitMQ 在集群节点上解压并重启的话，这样会抹去集群上的所有配置和数据。如果集群上的一切都能被重新创建的话，那就不会是什么问题。如果不是的话，那么升级着实是个复杂的过程。

首先你需要通过 RabbitMQ Management 插件使用第 6 章讲到的指令来备份当前配置。然后关闭所有生产者并等待消费者消费完队列中的所有消息（使用 rabbitmqctl 观察队列状态，直到所有队列都空了为止）[7]。现在，关闭节点，并解压新版本 RabbitMQ 到现有的安装目录。这时，选择其中一个磁盘节点作为升级节点。

---

6 由于RabbitMQ存储格式分别在1.x和2.0以及2.1.0和2.1.1有所更改，因此升级早于2.1.0的版本将会是一个手动升级的过程。如果你在更新的RabbitMQ版本之间拥有互不兼容的存储格式的话，RabbitMQ会自动将旧的存储文件复制到备份地址，并创建新的空白文件。

7 在RabbitMQ 2.6.0之后的新版本中，排空队列不是必需的，因为服务器会自动升级。不过这仍然是一个保障安全的好方法，以免在升级的时候出错。

当它启动的时候，该节点会将持久化的集群数据升级到新版本。然后你就能启动其他集群磁盘节点了，它们会获取升级后的集群数据。最后，启动集群内存节点，这样你就让集群上运行着闪亮的 RabbitMQ 新版本，而且所有的元数据和配置信息都会被保留。

在学习了升级和传统集群的操作后，现在是时候让我们看看如何对集群进行扩展了，以便能在节点失败时保留队列内容。

5.6 镜像队列和保留消息

在一开始讨论集群的时候，你也许会记得我们曾经说过默认情况下队列只存活于集群中的一个节点上。这仍然是对的，而且如果你使用的是 RabbitMQ 2.6.0 之前的任何一个版本的话，这也是唯一的选择。但是在 2.6.0 版本的时候，Rabbit 团队给我们带来了内建的双活冗余选项：镜像队列。像普通队列那样，镜像队列的主拷贝仅存在于一个节点（主队列，master）上，但与普通队列不同的是，镜像节点在集群中的其他节点上拥有从队列（slave）拷贝。一旦队列主节点不可用，最老的从队列将被选举为新的主队列。这听起来不就是在探索集群时我们一直寻找的高可用性“灵丹妙药”吗？不过有些注意事项你需要清楚。在我们深入探讨这些注意事项前，先看看如何编写消费者应用，对镜像队列加以利用。

5.6.1 声明并使用镜像队列

就如 AMQP 的许多方面一样，你的应用程序并不使用 `rabbitmqctl` 来定义镜像（mirrored）队列。声明镜像队列就像声明普通队列一样；你传入一个额外的 `x-ha-policy` 参数到 `queue.declare` 调用中。为了看看真实代码会是什么样子，让我们更新第 2 章写的那个 Hello World 消费者程序，让它声明镜像队列来替代普通的队列。将普通队列声明更改成镜像队列声明非常简单，只需将 `channel.queue_declare(queue="hello-queue")` 更改如下：

```
queue_args = {"x-ha-policy" : "all" }
channel.queue_declare(queue="hello-queue", arguments=queue_args)
```

`queue_args` 是简单的字典（或者哈希），包含了额外的队列声明参数。在该示例中，添加的参数为 `x-ha-policy`，并被设置为 `all`。当设置成 `all` 时，`x-ha-`

policy 告诉 Rabbit 你想让队列被镜像到集群中所有的节点上。这意味着如果在该队列声明之后，集群又新增节点的话，那么该节点就会自动托管一份队列的从拷贝。为了测试新的镜像队列消费者，我们从终端运行它，然后在另一个终端使用 rabbitmqctl 看看队列是否真的镜像了：

```
(terminal 1)> python hello_world_mirrored_queue_consumer.py

(terminal 2)> rabbitmqctl list_queues name pid slave_pids
Listing queues ...
hello-queue     <rabbit@Phantome.1.7429.1> [<rabbit2@Phantome.1.7...
...done.
```

当列出集群中的队列时，rabbitmq 会向你展示包括 name、pid 和 slave_pids 在内的属性。当处理镜像队列时，pid 是队列主拷贝的 Erlang 进程 ID，而 slave_pids 是其他节点上的从拷贝列表。在该示例中，在节点 rabbit@Phantome 有个 ID 为 1.7429.1 的主拷贝，还有在节点 rabbit2@Phantome 上的 ID 为 1.7431.1 的一个从拷贝。棒极了，你的 Hello World 消费者现在正在使用镜像队列！但是如果你不想让队列在集群的所有节点上都拥有镜像，而只想让它们之中的少数节点拥有的话，该怎么做呢？

这稍微有点棘手。由于队列的镜像性质是由应用程序在运行时指定的，RabbitMQ 团队决定你可以同样的方式在运行时指定队列需要镜像到哪些节点。因此，为了使队列镜像到集群中节点的子集中，你需要在程序中指明确切的节点名称（替代 all）。这是最棘手的部分，因为这意味着你必须在应用程序中硬编码。如果当应用程序尝试声明队列时，任何一个节点发生故障的话（或者从集群中移除的话），那么该声明就会失败。在 Rabbit 中并不存在可用组的概念，来将你想要让镜像队列存在的节点集合抽象成一个通用的名称，并且在运营团队决定对 Rabbit 集群重新布置后还能保持不变的。因此，我们强烈建议你尽可能使用 all 来设置 x-ha-policy。这样镜像队列就不用对声明进行硬编码来指明可能会变更的节点名称了。但是，假设你坚持想要为镜像队列指定集群节点的子集的话，该怎么做呢？

只需对镜像队列做两处更改，就能使用节点子集来替代集群所有节点的设置。首先，你需要把 x-ha-policy 从 all 更改为 nodes[8]。这样，RabbitMQ 就知道你会提供确切的集群节点名称列表。然后你向队列声明中添加另一个参数，叫作 x-ha-

8 在RabbitMQ的2.6.0和2.6.1版本中，该nodes队列复制分布策略并不起作用。在2.7.0及更新的版本中，它可以正常工作。

policy-params，并将其设置为队列想要镜像到的节点名称列表。代码如下所示：

```
queue_args = {"x-ha-policy" : "nodes",
              "x-ha-policy-params" : ["rabbit@Phantome"]}
channel.queue_declare(queue="hello-queue", arguments=queue_args)
```

如果你删除了在首次运行更新过的 Hello World 消费者创建的镜像队列（hello-queue），并重新运行了的话，你应该可以看到现在镜像队列只存在于集群中的单一节点上：

```
(terminal 1)> python hello_world_mirrored_queue_consumer.py

(terminal 2)> rabbitmqctl list_queues name pid slave_pids
Listing queues ...
hello-queue    <rabbit@Phantome.1.7429.1>    []
...done.
```

正如预期的那样，hello-queue 只有主拷贝而没有从拷贝（即便集群上有两个节点）。如果此时为队列添加新的从节点会怎样呢？新增的从拷贝只会包含那些在其添加进来之后从镜像队列发来的消息。RabbitMQ 不会（2.7.0 版本以前）将镜像队列现存的内容和新添加的从拷贝进行同步。理论上说消息是从现存的主拷贝和从拷贝上进行消费的，那些新增从拷贝不知道的所有旧消息会被移除，并且新增从拷贝最终会和现存的队列拷贝拥有相同的状态。但是如果在旧消息消费掉以前，你将所有包含现存的主拷贝和从拷贝的节点移除的话，那么新进的从拷贝会被推选为主拷贝，你会丢失那些旧消息。因此，直到 RabbitMQ 提供镜像队列现存内容到新从拷贝的同步机制前，分辨是否所有从拷贝都拥有相同的内容是很重要的。为了检测镜像队列的同步状态，在使用 rabbitmqctl 列出队列时添加 synchronised_slave_pids 即可：

```
> rabbitmqctl list_queues name pid slave_pids synchronised_slave_pids
Listing queues ...
hello-queue    <rabbit@Phantome.1.7429.1>    [<rabbit2@Phantome.1.74...
                                             [<rabbit2@Phantome.1.74...
...done.
```

如果第一个和第二个括号内的 PID 列表（即 [<rabbit2@Phantome.1.7431.1>]）是一模一样的话，那么所有的 slave 队列就都同步了。但是如果在第一个括号内的任何一个 PID 没有出现在第二个列表中的话，那就意味着缺失的从 PID 和稍早的从拷贝拥有不同的内容。如果是这种情况的话，那么你需要等到这两个括号中的列表内容完全相同之后，才能从集群中移除节点。这能确保只存在于你想要删除的节点

上的任何消息都不会丢失。这就是我们介绍镜像队列时提到的注意事项之一。为了更好地理解镜像队列的工作机制（以及剩下几项注意事项），让我们一起揭开镜像队列的神秘“面纱”。

到目前为止，你已经学习了集群的工作机制（包括镜像队列），而且不仅学会了如何在单台开发机器上部署，也学会了通过本地局域网络部署分布式集群。你甚至明白如何升级集群而永远不会丢失所有的配置。现在还差的是，在集群环境下如何编写代码以处理集群故障并自动重连到其他节点。

5.6.2 镜像队列工作原理

在非镜像队列的 Rabbit 集群中，信道负责将消息路由到合适的队列。当加入镜像队列后，信道仍然做着同样的事情。除了将消息按照路由绑定规则投递到合适的队列之外，它也要将消息投递到镜像队列的从拷贝（如图 5.5 所示）。在某种程度上，你可以将镜像队列视为拥有一个隐藏的 fanout 交换器，它指示着信道将消息分发到队列的从拷贝上。

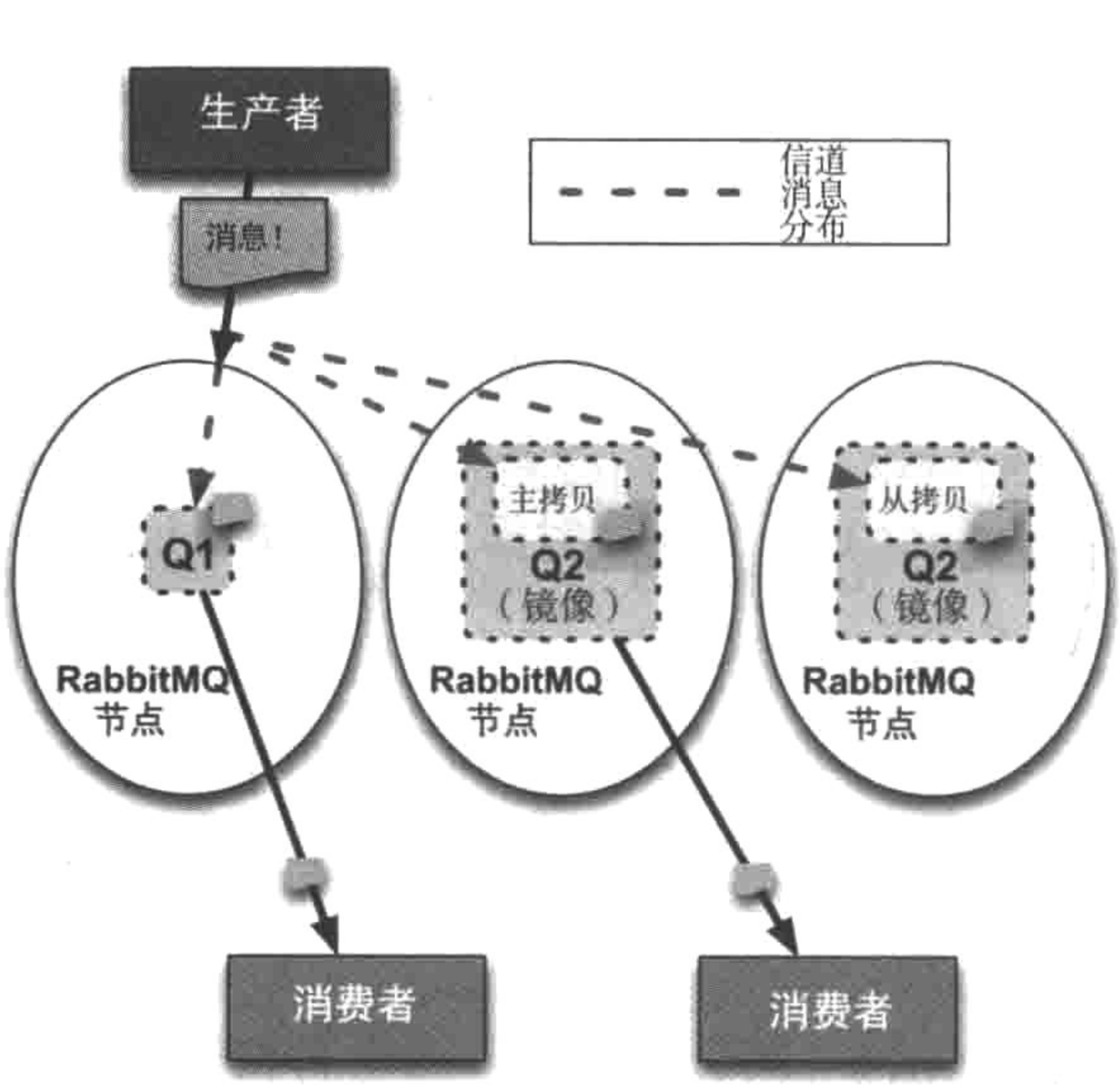

图 5.5 镜像队列行为

对于信道负责将消息并行地发布到镜像队列的主从拷贝上这一点的理解，有助于你理解镜像队列是如何影响事务和发送方确认模式的。当处理那些非镜像队列时，在信道根据匹配的绑定规则将消息路由到所有特定的队列之后，你只是收到一个发

送方确认消息（publisher confirm；或者一个成功的事务）而已。当切换到使用镜像队列时，Rabbit 使用相同的概念，只不过将概念拓展到了队列的从拷贝。所以如果你想要确认消息没有丢失的话，就可以对消息使用发送方确认模式，Rabbit 会在所有队列和队列的从拷贝安全地接收到消息时通知你。但是如果消息在路由到从拷贝前，镜像队列的主拷贝发生故障，并且该从拷贝变成了主拷贝的话，那么发送方确认消息永远不会到达，于是你就知道消息可能已经丢失了。不过，这只是关于发布方是如何处理镜像队列主节点故障的。那么附加在故障的主拷贝上的消费者会怎样呢?

如果镜像队列失去了一个从节点的话，则附加在镜像队列的任何消费者都不会注意到这一点。这是因为从技术上来讲，它们是附加在队列主拷贝上的。但是如果托管主拷贝的节点发生故障的话，那么所有该队列的消费者需要重新附加并监听新的队列主拷贝。对于通过故障节点进行连接的消费者来说，这倒没什么困难的。因为它们丢失了到节点的 TCP 连接，在它们重新附加到集群中一个新的节点时，会自动选取新的队列主拷贝。但对于那些通过节点附加到镜像队列且节点正常运行的消费者来说，RabbitMQ 会发送给这些消费者一个消费者取消（consumer cancellation）通知，告知它们已不再附加在队列主拷贝了。如果 AMQP 客户端库能够支持消费者取消通知的含义，那么它会抛出一个异常，同时应用程序就会知道已经不再附加到队列上了，需要重新附加。另一方面，如果你的客户端库不支持消费者取消的话，那事情就难办了。客户端无法通知应用程序其指向队列主拷贝的消费循环已不复存在。因此你的应用程序呆坐在那里，以为队列里没有消息可供消费。不幸的是，针对这种情况并没有好的解决方法（譬如，Rabbit 关闭消费者的信道以强制抛出异常）。所以如果客户端库不能支持消费者取消通知的话，你应该避免使用镜像队列。不然的话，你的队列会被未消费的消息塞满，然后监控系统就会在半夜用电话叫醒你。

在了解了取消通知之后，剩下唯一关于镜像队列需要注意的是，那些已经消费了但未被确认的消息。当镜像队列的主节点失败时，Rabbit 不得不对那些已经投递到消费者，但尚未得到确认的消息做个决定。虽然消息确实投递给了消费者，但是 Rabbit 却不能区分故障转移中丢失的确认消息和那些尚未得到确认的消息。所以为了安全起见，已经消费但尚未被确认的消息会重新入队到它们所原来在的队列位置（或者在队列的末尾，在版本 2.7.0 之前）。

5.7 总结

当开始本章之前，你完全受制于使用单一 Rabbit 节点来为应用程序提供动力和通信。现在你不再担心 Rabbit 崩溃了，因为 RabbitMQ 集群会确保基础架构正常运作。更重要的是，你理解了集群内部实现的细枝末节，所以你能为 RabbitMQ 架构做出明智的设计决定。这些决定可以最大化正常运行时间和可扩展性，并同时最小化消息丢失带来的负面影响。但是 RabbitMQ 集群只是所有工作的一半。即便拥有动力十足的 Rabbit 集群作为基础架构的“心脏”，你的应用程序仍然只是连接到了一个节点。如果该集群节点发生故障的话，客户端会在集群中进行重连，这样它们就可以不用停顿，继续运作。集群是 Rabbit 高可用性的最终方案。是时候讨论如何编写可以幸免于单一节点故障的应用程序代码了，以便为这个高可用性方案画上圆满的句号。

6 从故障中恢复

本章要点

- 理解负载均衡
- 安装并配置HAProxy来为Rabbit做负载均衡
- 重连并从故障中恢复

构建 RabbitMQ 集群来确保可用性和性能只是保障弹性消息通信基础架构的一半。另一半则是编写当节点发生故障时知道如何重连到集群的应用程序。处理到集群的重连有多种策略，我们所关注的这种是使用负载均衡来处理节点的选择。通过使用负载均衡，不仅可以减少应用程序处理节点故障代码的复杂性，又能确保在集群中连接的平均分布。但是即便使用了负载均衡器，编写处理节点故障的应用程序也要比建立到集群的连接复杂得多。应用程序需要准备好重新创建交换器和队列以应对初始节点故障。当使用主备设置的两台独立 Rabbit 节点时更为重要（我们会在第 7 章介绍）。在开始编写故障处理代码之前，我们先看看 RabbitMQ 中如何使用负载均衡器。

6.1 为Rabbit做负载均衡

限于每个人不同的背景，你也许并不了解负载均衡器（load balancer）。一台负载均衡器表示了单个 IP 地址背后存在多台服务器。假设你拥有三台服务器来为网站提供支撑，它们分别是 web1.acme.com、web2.acme.com 和 web3.acme.com。如果没有负载均衡器的话，你的客户不得不手工指定访问 web1，并且如果 web1 宕机的话，再尝试 web2 或者 web3。这不仅给你的客户带来了麻烦，同时又让你对每台服务器的负载失去控制。如果拥有负载均衡器的话，你可以为其创建一个 IP 地址，命名为 www.acme.com。然后当客户连接到 www 时，负载均衡器会根据负载情况透明地将连接代理到 web1、web2 或者 web3。如果 web1 宕机了，负载均衡器会足够聪明地检测到这一点，并停止将连接发送到 web1，转而发送至 web2 和 web3。对你的客户来说，这就像你拥有一台巨大的叫作 www 的服务器，他们只需关心这一点就够了。所有的负载均衡和故障服务器检测都交由负载均衡器一手包办（见图 6.1）。

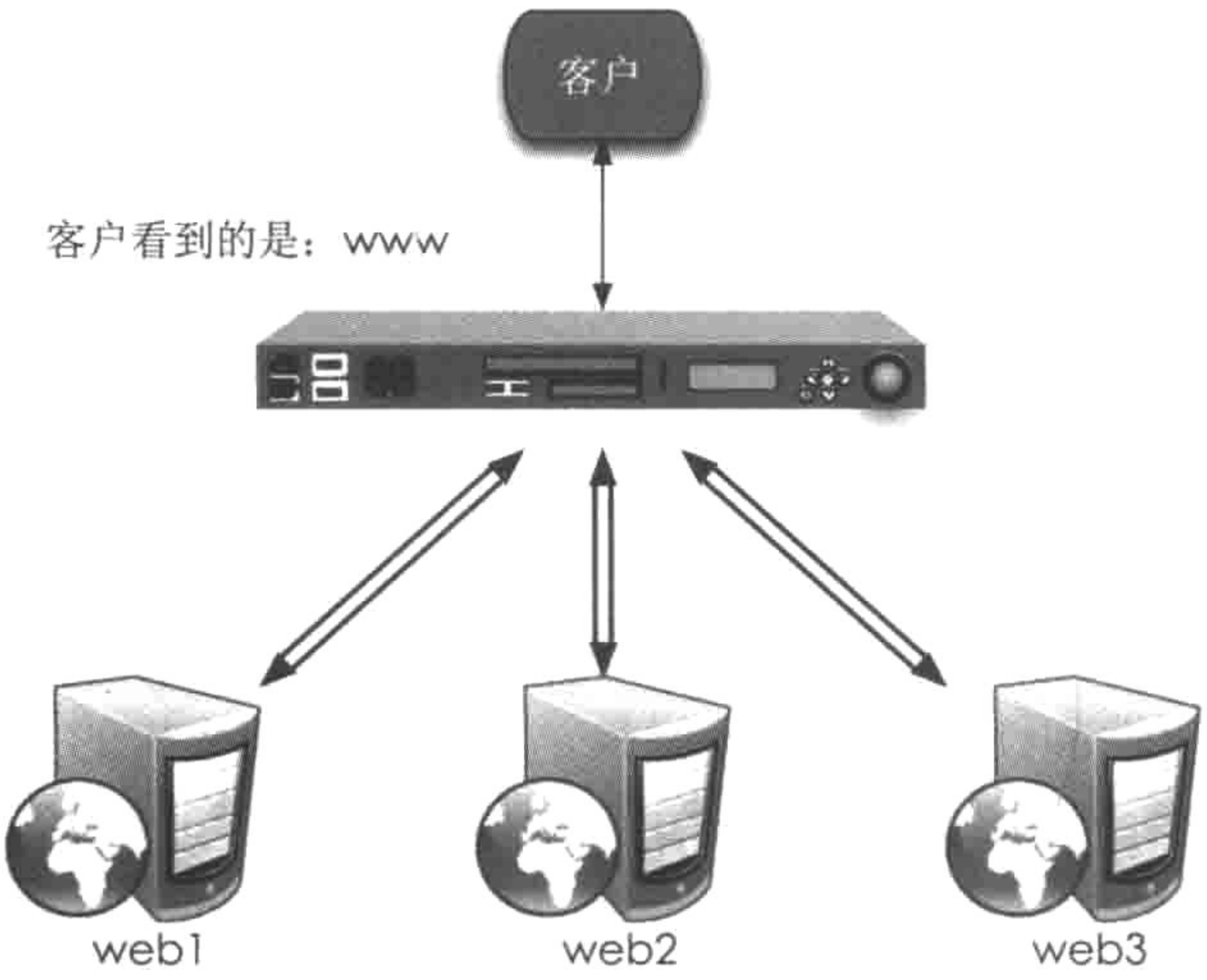

图 6.1 将多台 web 服务器做负载均衡，看起来像一台服务器一样

当为 RabbitMQ 添加负载均衡器时，集群节点就作为负载均衡器背后的服务器，而你的生产者和消费者就是客户了。应用程序只需知道负载均衡器的前端 IP；负载均衡器会以最小的连接负载透明地将客户端连接到集群节点。如果你想要为集群增强性能，则无须更改应用程序——你只需服务器新增节点，然后将其添加到负载均衡器配置即可。无须修改应用程序。如果没有负载均衡器的话，应用程序就不得不对每一个集群节点的 IP 做手工配置。同时也不得不自行处理集群节点的选择和故障

服务器检测。由于每个应用程序需要自行选择节点，因此你就无法在集群内将负载平均分配。负载分配基本上是随机的。所以通过将负载均衡器放置在 Rabbit 集群的前端，你就可以让它来处理节点选择、故障服务器检测以及负载分布这些复杂的事情了（见图 6.2）。

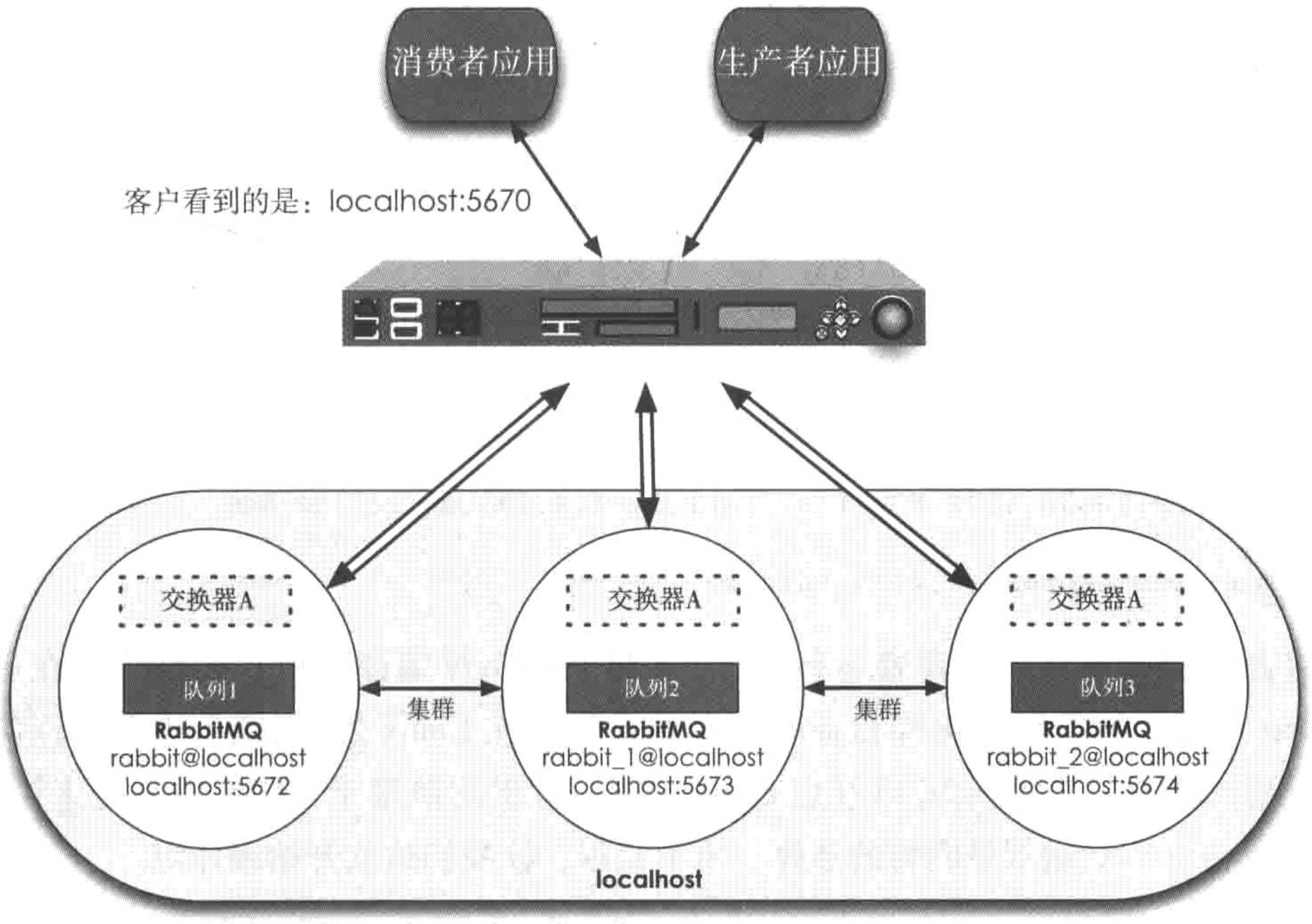

图 6.2 为 RabbitMQ 集群做负载均衡

市面上有许多不同的负载均衡器，它们可以分为两类：硬件装置和软件。硬件装置是非常强大的专用网络系统，可以轻松处理每秒数以百万计的连接。它们经常提供高级集群设置，使得两个负载均衡器能够像一个独立单元那样，以应对单点失败。如果你已经有了一个硬件负载均衡器的话，那就使用它来作为 RabbitMQ 集群的负载均衡，将其设置成第四层负载均衡模式。不过绝大多数情况下，一个软件负载均衡器就已经绰绰有余了。在超过软件负载均衡器所能满足的集群能力之前，你更有可能先达到集群能够支持的最大节点数上限。在众多的软件负载均衡器里面，我们选择使用 HAProxy。它是免费的，而且非常可靠，并且为各种站点处理高负载，例如 StackOverflow。同时，它可以运行在几乎所有的基于 UNIX 的平台上，并且非常容易配置。对 HAProxy 的夸赞到此为止，下面展示如何安装它并进行配置，为

RabbitMQ 集群做负载均衡。

6.1.1 安装 HAProxy

安装 HAProxy 非常简单。大多数现代 Linux 发行版的包管理系统都有现成的 HAProxy。不过，我们会从源代码编译构建。首先将 HAProxy 下载到你的开发系统中，并解压缩：

```
$ wget http://haproxy.1wt.eu/download/1.4/src/haproxy-1.4.14.tar.gz
Resolving haproxy.1wt.eu... 88.191.124.161
Connecting to haproxy.1wt.eu|88.191.124.161|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 812238 (793K) [application/x-gzip]
Saving to: `haproxy-1.4.14.tar.gz'

100%[==============================>] 812,238     192K/s  in 4.1s

2011-03-29 19:16:56 (192 KB/s) - `haproxy-1.4.14.tar.gz' saved

$ sudo tar xfz haproxy-1.4.14.tar.gz
```

将源代码解压之后，需要运行 make 来将 HAProxy 编译为可执行程序。在运行 make 之前，你需要先选择目标平台。如果你是在 Linux 2.6 系统上进行编译的话，设置 TARGET=linux26 以开启 epoll 支持。对于其他基于 UNIX 的系统来说，TARGET=generic 通常是明智的选择。事不宜迟，让我们这就开始编译吧。

> **注意**：epoll 是 Linux 2.6 内核的一个功能，它使得像 HAProxy 这样基于事件的网络软件在新的包到达时收到通知，以取代周期轮询。这有助于减少 CPU 的占用并在负载较重的系统中提升性能。FreeBSD 也有相似的功能，叫作 kqueues。在 HAProxy 中通过设置 TARGET=freebsd 来加以利用。

```
$ cd haproxy-1.4.14
$ sudo make TARGET=generic
...
gcc  -g -o haproxy src/haproxy.o src/sessionhash.o src/b...
```

现在在 build 目录下应该有个名为 haproxy 的可执行文件了。如果一切正常，你应该可以运行 haproxy --help 来看看它的配置选项。最后，将 haproxy 可执行文件复制到 /usr/local/sbin，这样就可以在 UNIX 路径下可用了。现在需要对 HAProxy 进行配置，这样它就可以在开发系统上为 RabbitMQ 集群做负载均衡了。

6.1.2 配置 HAProxy

HAProxy 使用单一配置文件来定义所有属性，包括从前端 IP 到后端服务器。下列清单展示了用于本地 Rabbit 集群负载均衡的配置。

清单 6.1 本地 RabbitMQ 集群的 HAProxy 配置

```
global
    log 127.0.0.1    local0 info
    maxconn 4096
    stats socket /tmp/haproxy.socket uid haproxy mode 770 level admin
    daemon

defaults
    log    global
    mode    tcp
    option    tcplog
    option    dontlognull
    retries    3
    option redispatch
    maxconn    2000
    timeout connect    5s
    timeout client 120s
    timeout server 120s

listen rabbitmq_local_cluster 127.0.0.1:5670
    mode tcp
    balance roundrobin
    server rabbit 127.0.0.1:5672 check inter 5000 rise 2 fall 3
    server rabbit_1 127.0.0.1:5673 check inter 5000 rise 2 fall 3
    server rabbit_2 127.0.0.1:5674 check inter 5000 rise 2 fall 3

listen private_monitoring :8100
    mode http
    option httplog
    stats enable
    stats uri   /stats
    stats refresh 5s
```

日志选项

加载默认负载均衡配置

❶ 前端IP，供消费者和生产者使用

❷ 负载均衡选项

❸ 负载均衡中的集群节点配置

❹ 数据统计页面

在❶处，你定义了 IP 和端口以供客户端连接。你使用了 5670 端口，这样不会和 RabbitMQ 集群节点相冲突。然后你告诉 HAProxy 使用轮询算法❷来将负载分发给后台服务器（见图 6.3）。

最有趣的部分是❸，在这里我们定义了后台服务器：

```
server rabbit 127.0.0.1:5672 check inter 5000 rise 2 fall 3
    server rabbit_1 127.0.0.1:5673 check inter 5000 rise 2 fall 3
    server rabbit_2 127.0.0.1:5674 check inter 5000 rise 2 fall 3
```

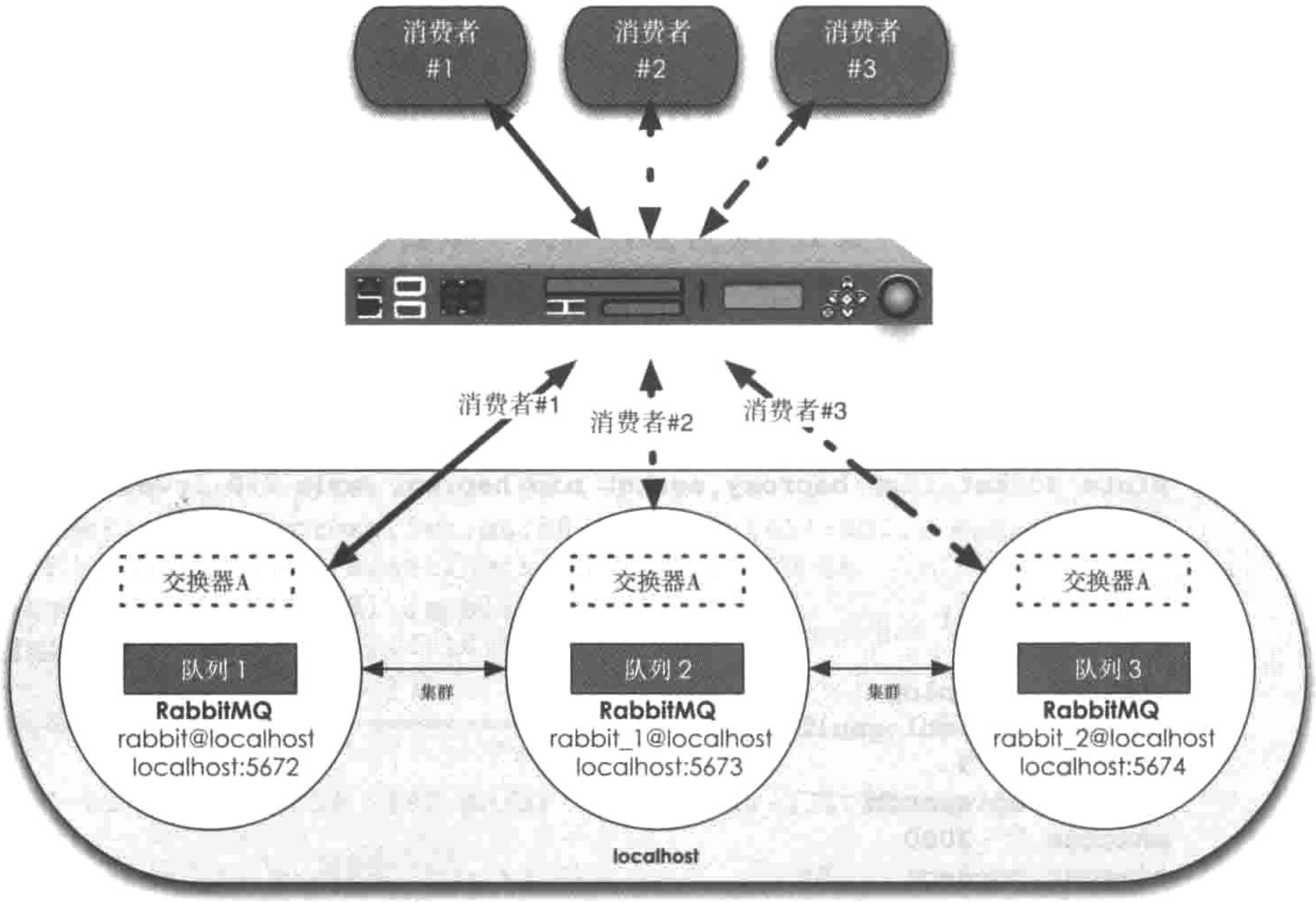

图 6.3 轮询负载均衡

每台后台服务器定义指令包含五个部分。

- `server <name>`：后台服务器定义中的内部标识。
- `<IP>:<port>`：连接到后台服务器的 IP 和端口号。
- `check inter <value>`：定义了每隔多少毫秒检查后台服务器是否可用。
- `rise <value>`：表明了后台服务器在发生故障之后，需要完成多少次健康检测，才能再次被确认可用。
- `fall <value>`：表明需要经历多少次失败的健康检测，HAProxy 才会停止使用后台服务器。

最后的配置部分是数据统计页面❹。这是可选的，可以通过端口 8100 启用它，你可以连接到 http://localhost:8100 观测 HAProxy 的当前状态，包括当前每台 RabbitMQ 集群节点正在服务多少条连接。当你想要查看集群负载情况或者当前有多少节点在工作的时候是很有用的。还有很多 HAProxy 配置选项，涵盖了方方面面，包括了从复杂的负载均衡规则，到将后台节点视为备用服务器（仅在所有主服务器发生故障时才启用）。在 http://haproxy.1wt.eu/download/1.4/doc/configuration.txt 上有

HAProxy 的手册以供查阅。

让我们用新的配置来启动 HAProxy 并确保它能工作。运行 `/usr/local/sbin/haproxy -f config_file`。其中的 config_file 就是你刚刚创建的配置文件。如果一切正常的话，你应该可以顺利加载 http://localhost:8100/stats 页面，它看起来像图 6.4 那样。

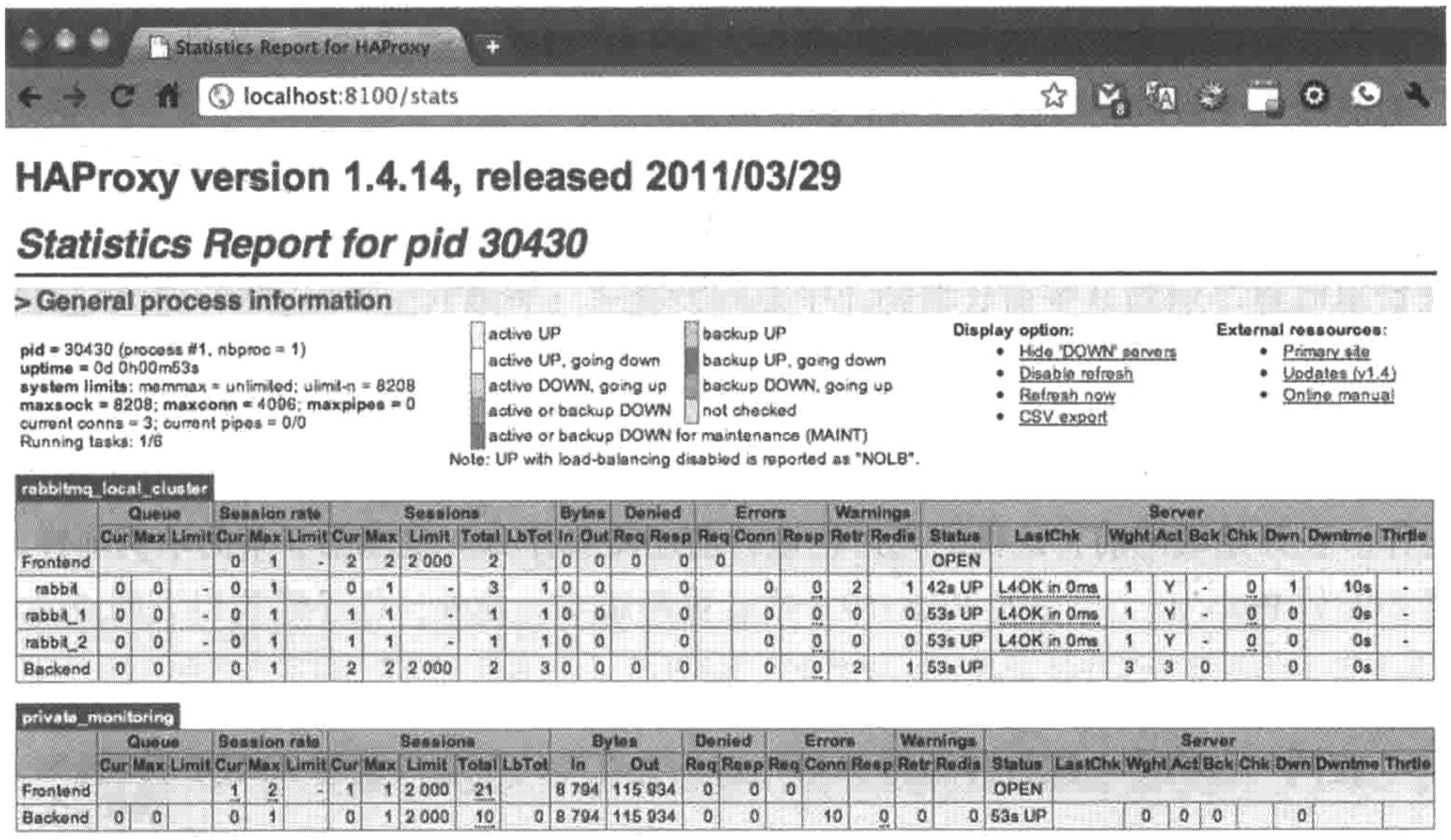

图 6.4 HAProxy 数据统计页面

现在开发系统上运行着负载均衡器，我们准备深入探索如何使用它来为消息通信应用程序植入故障转移和快速恢复的能力。

6.2 连接丢失和故障转移

当集群节点出现故障时，应用程序必须要做出决定：下一个该连向哪里？为了能有效回答这个问题，你必须在事情发生前就有所准备。优雅地处理节点故障需要思维的转变。集群并不意味着可以完全避免所有 Rabbit 问题；而是意味着当出现问题的时候节点能继续运行。所以，首先需要退一步思考一下，在编写代码之前你可以做怎样的假设：

（1）如果我重新连接到新的服务器，那么我的信道以及其上的所有消费循环会

怎样呢？它们现在都失效了。你必须对它们进行重建。

（2）当我进行重连的时候，我能否假设所有的交换器、队列和绑定仍然存在于集群之中？我能否重连之后立即开始从队列消费呢？答案是否定的。你不能假设队列和绑定可以从节点故障中恢复。你必须假定你在消费的所有队列都附加在该节点之上——并且已不复存在。对队列的绑定来说也一样，然而交换器则不同。如果你使用的是内建的 Rabbit 集群的话，则可以假设交换器能够幸免于节点故障，因为它们在所有节点都有副本。但是如果你使用了我们在后面所讲述的主 / 备模式设置的话，则仍然无法假设交换器可以从故障中恢复。

从这些问题中你可以得出的结论是当故障转移到新的节点时，你无法对集群的状态做任何假定。虽然 Rabbit 集群可以让你重连到新节点去，但你不能做任何假设。从某些角度而言，你应该总是将故障转移视为连接到了一个完全无关的 RabbitMQ 服务器，而不是有着共享状态的集群节点。因此，不论节点故障什么时候发生，在检测到故障并进行重连之后的首要任务是构造交换器、队列和绑定，以便应用程序的运作。在开始编码前，让我们先讨论运行前还需要什么。就像第 3 章的示例那样，你需要安装 Python 2.6，以及 Pika 0.9.6 或者更高版本。此外，你还需要以下两点：

- 本地开发机器设置了 RabbitMQ 集群。
- 在同一台机器上配置并运行 HAProxy，并在 5670 端口监听 AMQP 连接。

你将编写能够从集群节点故障中恢复的生产者和消费者示例。构造一个集群感知的消费者更为困难，因为是由消费者来搭建消息通信结构的（交换器、队列和绑定）。因此，这就取决于消费者在节点发生故障之后重建通信结构了。标准的消费者代码如下列清单所示。

清单 6.2 标准消费者主体

```
conn_broker = pika.BlockingConnection(conn_params)

channel = conn_broker.channel()
channel.exchange_declare( exchange="cluster_test",
                          type="direct",
                          auto_delete=False)
channel.queue_declare( queue="cluster_test",
                       auto_delete=False)
channel.queue_bind( queue="cluster_test",
                    exchange="cluster_test",
                    routing_key="cluster_test")
```

❶ 连接到 RabbitMQ

❷ 自定义连接行为

❸ 声明交换器、队列和绑定

```
print "Ready for testing!"
channel.basic_consume( msg_rcvd,
                       queue="cluster_test",
                       no_ack=False,
                       consumer_tag="cluster_test")
channel.start_consuming()
```

❹ 开始消费消息

在❶处，你使用已经构造好的参数来连接服务器。你建立了信道❷，并开始声明❸交换器、队列和绑定（消息通信结构）。在消息通信结构构造完成之后，你创建了❹消费订阅（由 msg_rcvd 函数提供），并开始消费消息。这时，如果遇到了节点故障，那么程序就会由于未处理的异常而崩溃。这是因为代码不知道在连接发生错误时该怎么办。你需要做的是将这段代码包裹在一个异常处理中，并在故障发生时发起重连。但是从哪里开始呢？主要代码块中的哪部分需要在故障后重新运行呢？答案是整个代码都需要。如果你假设消息通信结构的任何部分都无法从节点故障中恢复的话，那么就需要在每次错误发生的时候运行整个代码块。记住这一点，然后重写主代码块，那么就会像下列清单所示。

清单 6.3 集群感知的消费者主体

```
while True:
    try:
        conn_broker = pika.BlockingConnection(conn_params)
        channel = conn_broker.channel()
        channel.exchange_declare( exchange="cluster_test",
                                  type="direct",
                                  auto_delete=False)
        channel.queue_declare( queue="cluster_test",
                               auto_delete=False)
        channel.queue_bind( queue="cluster_test",
                            exchange="cluster_test",
                            routing_key="cluster_test")

print "Ready for testing!"
channel.basic_consume( msg_rcvd,
                       queue="cluster_test",
                       no_ack=False,
                       consumer_tag="cluster_test")
channel.start_consuming()
except Exception, e:
   traceback.print_exc()
```

默认情况下，重新连接到 RabbitMQ ❶

❷ 建立到 RabbitMQ 的连接

❸ 声明交换器、队列和绑定

❹ 开始消费消息

❺ 捕获连接异常，并进行打印

通过将主代码块放入 try...except 块中，你就可以检测连接故障❺，消费者也不会因此而崩溃。在该示例中，你捕获了所有的错误并将它们打印到屏幕上。但这只是解决方案的一半。虽然当节点发生故障时程序不再崩溃了，但你仍然需要进行重连并重建通信结构。为了实现这一点，你需要将整个代码主体（包括 try...except 块）包含在无限循环之中❶。当应用程序第一次启动时，它进入循环体，然后构造连接❷和通信结构❸。之后它暂停循环开始消费❺。只要没有错误发生，外层循环就会一直运行下去。但当节点发生故障时，就会抛出连接错误。这会导致控制权从消费代码❹转移到外层异常处理❺。接着，异常处理代码通过捕获错误来阻止程序崩溃，并将错误打印到屏幕，随后把控制权交还给外层循环。现在循环重新启动整个连接过程❶，像之前那样构造新的连接❷和通信结构❸。这个简单的更改却让程序能够处理 RabbitMQ 集群下的节点故障。同时也能处理节点故障的各种变体。这一点我们尚未讨论过，但是你必须知道。

到目前为止，在我们从应用程序的角度讨论集群节点故障时，我们反复说明应用程序连接的是节点。你也许会做这样的假设：只要应用程序连接的那个节点没有崩溃，那么就没什么好担心的了。这并非完全正确。如果你记得队列在集群环境下的运作方式的话，就会注意到它们只存在于某一个节点上。由于在开始消费的时候，应用程序并不知道队列在哪个节点上，因此，应用程序很有可能连接到了集群中的 A 节点但却从 B 节点的队列上消费消息。所以，当 B 节点发生故障时会发生什么呢？虽然应用程序不会遭受连接错误，但是消费的那个队列却已不复存在。如果你运行的是 RabbitMQ 2.4.0 之前的版本，那你的运气就太糟了。消费者会呆呆坐在那儿很开心，但却什么事情都不做（至少在你重启之前）。这个限制更多的是由于 AMQP 造成的。但在 RabbitMQ 2.4.0 时，它为我们带来了 AMQP 的一个新的扩展，叫作取消通知（cancellation notification）。通过取消通知，除了自行取消之外的任何情况导致的订阅终结，消费者都会收到通知。在 Pika 中，这种模式会表现为消费代码抛出的异常。这会被异常处理代码捕获，然后重连并重建通信结构。将所有这一切结合起来，消费者代码就会如下列清单所示。

清单 6.4 集群感知的消费者

```
import sys, json, pika, time, traceback

def msg_rcvd(channel, method, header, body):
    message = json.loads(body)

    print "Received: %(content)s/%(time)d" % message
    channel.basic_ack(delivery_tag=method.delivery_tag)

if __name__ == "__main__":
    AMQP_SERVER = sys.argv[1]
    AMQP_PORT = int(sys.argv[2])

    creds_broker = pika.PlainCredentials("guest", "guest")
    conn_params = pika.ConnectionParameters( AMQP_SERVER,
                                             port=AMQP_PORT,
                                             virtual_host="/",
                                             credentials=creds_broker)

    while True:
        try:
            conn_broker = pika.BlockingConnection(conn_params)

            channel = conn_broker.channel()
            channel.exchange_declare( exchange="cluster_test",
                                      type="direct",
                                      auto_delete=False)
            channel.queue_declare( queue="cluster_test",
                                   auto_delete=False)
            channel.queue_bind( queue="cluster_test",
                                exchange="cluster_test",
                                routing_key="cluster_test")

            print "Ready for testing!"
            channel.basic_consume( msg_rcvd,
                                   queue="cluster_test",
                                   no_ack=False,
                                   consumer_tag="cluster_test")
            channel.start_consuming()
        except Exception, e:
            traceback.print_exc()
```

打印并发送确认消息

服务器设定

建立到服务器的连接

故障发生时，重新连接 RabbitMQ

建立到 RabbitMQ 的连接

自定义连接行为

声明交换器、队列和绑定

开始消费消息

捕获连接错误并打印出来

如你所见，将任何消费者应用转换为集群感知的并不困难。只需要对在节点发生故障时 RabbitMQ 内部运作有所了解，并将那些应对措施融入到代码之中。现在就让我们启动消费者程序看看会发生什么：

```
$ python cluster_test_consumer.py localhost 5670
Ready for testing!
```

你通过负载均衡器连上了集群，并搭建了通信结构（访问 http://localhost:8100 以观察你连接上的是哪一个集群节点）。但是，你的消费者并没有做什么有趣的事情。你需要让集群生产者给消费者发送一些内容以做显示！生产者只是短暂运作，所以不需要任何花哨的故障处理代码。这是因为生产者的每次调用都会重新建立新的连接，这使得负载均衡器可以选择新的功能节点。以下清单展示了生产者示例。

清单 6.5 集群感知的生产者

```
import sys, time, json, pika

AMQP_HOST = sys.argv[1]
AMQP_PORT = int(sys.argv[2])

creds_broker = pika.PlainCredentials("guest", "guest")
conn_params = pika.ConnectionParameters(AMQP_HOST,
                                        port=AMQP_PORT,
                                        virtual_host = "/",
                                        credentials = creds_broker)

conn_broker = pika.BlockingConnection(conn_params)

channel = conn_broker.channel()


msg = json.dumps({"content": "Cluster Test!",
                  "time" : time.time()})
msg_props = pika.BasicProperties(content_type="application/json")

channel.basic_publish(body=msg,
                      exchange="cluster_test",
                      properties=msg_props,
                      routing_key="cluster_test")

print "Sent cluster test message."
```

❶ 建立到代理服务器的连接

❷ 连接到 RabbitMQ 并发送消息

生产者代码的第一部分❶像之前那样设置了连接。根据传给生产者的命令行参数来确定 RabbitMQ“服务器”的 IP 地址（第一个参数）和端口（第二个参数）。这里使用的是每次 Rabbit 安装后内建的 guest 账户做认证。然后你创建了一条 JSON 消息❷发送给消费者，内容包含了 `Cluster Test!` 和当前时间戳。在发送消息之前你设置了消息的 `content_type` 头，这样消费者就知道这条消息是 JSON 编码的。最后，你将消息通过 Rabbit 发送给消费者。那么命令行上是怎样的呢？

```
$ python cluster_test_producer.py localhost 5670
Sent cluster test message.
```

回过头检查一下运行的消费者……

```
$ python cluster_test_consumer.py localhost 5670
Ready for testing!
Received: Cluster Test!/1301531152
```

太棒了！生产者从集群的一个节点发送消息，同时消费者从另一个节点接收消息！让我们马上亲身实践，重启消费者连接的节点。首先，使用 http://localhost:8100 找出消费者连接的集群节点名称。它应该是在 HAProxy 数据统计页面里的 Sessions 列组下的 Cur 列内以 1 列出的节点。然后从 RabbitMQ 安装目录下运行 ./sbin/rabbitmqctl -n node_name stop_app，这里的 node_name 是你在 HAProxy 数据统计里确认的节点名称。现在回过头来看一下消费者屏幕，你应该看到一些连接错误，在这之后成功连接到集群。

```
$ python cluster_test_consumer.py localhost 5670
Ready for testing!
Traceback (most recent call last):
  File "cluster_test_consumer.py", line 57, in <module>
    channel.start_consuming()
  File "/Library/Python/2.6/site-packages/pika-0.9.6-py2.6.egg/
  pika/adapters/blocking_connection.py", line 293, in start_consuming
    self.transport.connection.process_data_events()
  File "/Library/Python/2.6/site-packages/pika-0.9.6-py2.6.egg/
  pika/adapters/blocking_connection.py", line 87, in process_data_events
    raise AMQPConnectionError
AMQPConnectionError
Ready for testing!
```

第二行 Ready for testing! 表示消费者已经从节点故障中成功恢复并重新连接到了集群。现在如果你向集群发布一条新的消息，则应该可以从重新连接的消费者处看到回显。

```
$ python cluster_test_consumer.py localhost 5670
Ready for testing!
Traceback (most recent call last):
 File "cluster_test_consumer.py", line 57, in <module>
   channel.start_consuming()
 File "/Library/Python/2.6/site-packages/pika-0.9.6-py2.6.egg/
 pika/adapters/blocking_connection.py", line 293, in start_consuming
   self.transport.connection.process_data_events()
 File "/Library/Python/2.6/site-packages/pika-0.9.6-py2.6.egg/
 pika/adapters/blocking_connection.py", line 87, in process_data_events
   raise AMQPConnectionError
AMQPConnectionError
Ready for testing!
```

```
Received: Cluster Test!/1301531677
```

一切工作正常！现在你拥有了一个完整的 RabbitMQ 集群设置，包括一个负载均衡器用来处理节点选择，以及兼容集群的消费者和生产者（能在集群节点发生故障时保持通信）。现在你终于可以满意地收工了，因为你已经学会了 RabbitMQ 的内建集群了。但是这个内建集群并不能覆盖所有的使用场景。举个例子，当节点发生故障时，要是你绝对无法承担丢失任何持久化队列消息的风险该怎么办呢？当前的 RabbitMQ 版本会在节点重新加入集群时，恢复持久化队列和其内容，但是 1.8.0 之前的版本却不行。这需要一个独立运行的主 / 备机 RabbitMQ 服务器替代方案。它允许你在不丢失老队列内容的基础之上进行故障转移。

6.3 总结

RabbitMQ 集群只是构建快速恢复的消息通信基础架构的一半。另一半取决于你的应用程序。现在你知道了如何编写程序，可以在面对集群节点故障时快速恢复：那就是通过重新连接新的节点并重建通信结构以使得程序继续运作。同样重要的是，你现在可以设置并使用负载均衡器来确定哪个集群节点有故障，同时在应用程序重连时智能地路由到新的节点。这些技术和 RabbitMQ 集群相结合，给你带来了健壮的消息通信基础架构而不会受到故障的影响，也不会让你的应用程序停顿一下。但是仍然有一些关于 Rabbit 高度可用性的悬而未决的问题。比如，你如何设计 Rabbit 基础架构使得当节点发生故障时，持久化队列仍然可用？同样地，如何设计 Rabbit 架构使得可以避免失去整个数据中心和集群？要知道这些问题的答案，我们需要拿出“铲子”（Shovel）跳入“兔穴”（warren）之中。

7 warren和Shovel：故障转移和复制

本章要点

- 理解主/备方式（warren）
- 使用负载均衡器创建warren
- 使用Shovel构建远距离复制

到目前为止，当我们讨论高可用性的时候，总是与 RabbitMQ 内建的集群有关。但集群并非是向 Rabbit 基础架构里植入弹性的唯一方法。而且根据你的需求，这也并不总是一个正确的方法。集群迫使你不得不在以下两者之间做权衡：所有节点表现得像独立单元来分布负载的优点，但是在故障节点恢复前无法使用可持久化队列的缺点。同时，集群无法满足你构建分布在多个数据中心的 RabbitMQ 架构。因此，虽然集群最初听起来像是解决可用性问题的“瑞士军刀”，但是你仍然需要其他工具。这正是 warren 和 Shovel 的用武之地。

利用到目前为止所掌握的知识，你将会学习如何构建主 / 备机的独立 RabbitMQ 服务器，当面对持久化消息通信时，让你可以用扩展性换取灵活性。然后你会发现你能使用 Shovel 插件将一座城市的 Rabbit 服务器上的队列内容复制到遥远的另一座城市的 Rabbit 服务器（或者集群）。当你完成这些内容的时候，你将拥有一个完整

的工具箱来应对任何高可用性场景。最棒的是，如果你使用从第 12 章中习得的新技术来设计应用程序的话，你就能不做更改地使用它们来利用这些新的 RabbitMQ 拓扑。让我们开始深入理解主 / 备机模式，也就是 warren 模式吧。

7.1 warren：另一种集群方式

在 RabbitMQ 1.8.0 之前的版本中，当包含持久化队列的集群节点发生故障时，会发生“非常有趣”的行为。当节点仍处于故障中时，如果客户端重新创建了持久化队列，那么当节点恢复时，旧队列里的内容就会丢失。恢复了的节点应该会说：“哎呀，这个队列已经存在了；我不需要我的拷贝。”当旧的队列内容是有价值的时候，那问题就大了。从 1.8.0 版本开始，这个行为发生了变化。当拥有持久化队列的节点发生故障时，该队列无法被重新创建，任何尝试重新声明队列的客户端都会收到一个 404 NOT_FOUND AMQP 错误。当故障节点恢复时，持久化队列及其内容也跟着恢复了（假设消息是以 delivery_mode 2 的模式进行投递的）。但是在节点恢复以前，任何应该投递到该队列的消息要么丢失了，要么由于设置了 mandatory 发布标志导致客户端收到了错误。

如果应用程序无法承担丢失消息的风险，或者在故障队列恢复前无法处理不断重发消息的延迟，那么你需要我们提到的 warren 模式。用我们的话来说，一个 warren 是指一对主 / 备独立服务器，并前置一台负载均衡器来处理故障转移（见图 7.1）。这样设置的好处是，这是真正的无共享架构。主服务器和备用服务器之间没有协作，所以任何影响到主服务器的问题不会自动转移到备用服务器上；反过来也一样。它们之间分隔得足够彻底，以至于你可以运行不同版本的 RabbitMQ。这样允许你在生产环境中引进新版本的 RabbitMQ，同时保有旧版本以防万一。拥有两台完全隔离的 RabbitMQ 服务器来处理彼此的负载没什么不好的。不管什么原因，当 Rabbit 的内建集群无法满足需求的时候，warren 就能派上用场。

还可以用其他方式设置 warren 来达到高可用性。一个不同思想流派谈道：“当主节点发生故障时，我想让备用节点拥有主节点的所有消息。”我们的负载均衡器和无共享架构无法做到这一点。取而代之的是我们的方法给了你一个可以马上开始重新发布和消费消息的地方；同时当主节点恢复时，它允许你的消费者重新连接并消费主节点发生故障时队列上的那些消息。你不会丢失任何老的或者新的消息，但是你不得不等待主节点恢复之后旧消息才能重新变为可用。针对构建 warren，另一流

派表达了不同的看法，他们认为你应该在主服务器和备用服务器之间使用共享存储的方式来搭建 warren，备用节点上的 RabbitMQ 不用运行（见图 7.2）。那么当主服务器发生故障时，你使用 Pacemaker[1] 将 RabbitMQ 的 IP 地址转移到备用节点上并启动 Rabbit 服务器，采用共享存储上的当前元数据、内容和状态。我们认为这种设置方式有几个问题。首先一点，因为存储是共享的，所以如果由于某些原因导致主节点发生故障的话，那么这种错误同样会在备用节点上发生，并会阻止 RabbitMQ 的正常启动。第二点，你需要确保备用 RabbitMQ 和主节点上的 RabbitMQ 拥有相同的节点名称和 UID。如果这两处有一处不同，备用 Rabbit 将无法访问共享存储上的文件，也就无法启动了。最后，使用这种配置的 warren 意味着备用 Rabbit 并没有运行。因此有可能存在某些情况更改了备用节点，以致在你需要的时候 Rabbit 无法启动。这些都太过复杂了，我们当然选择简单的。

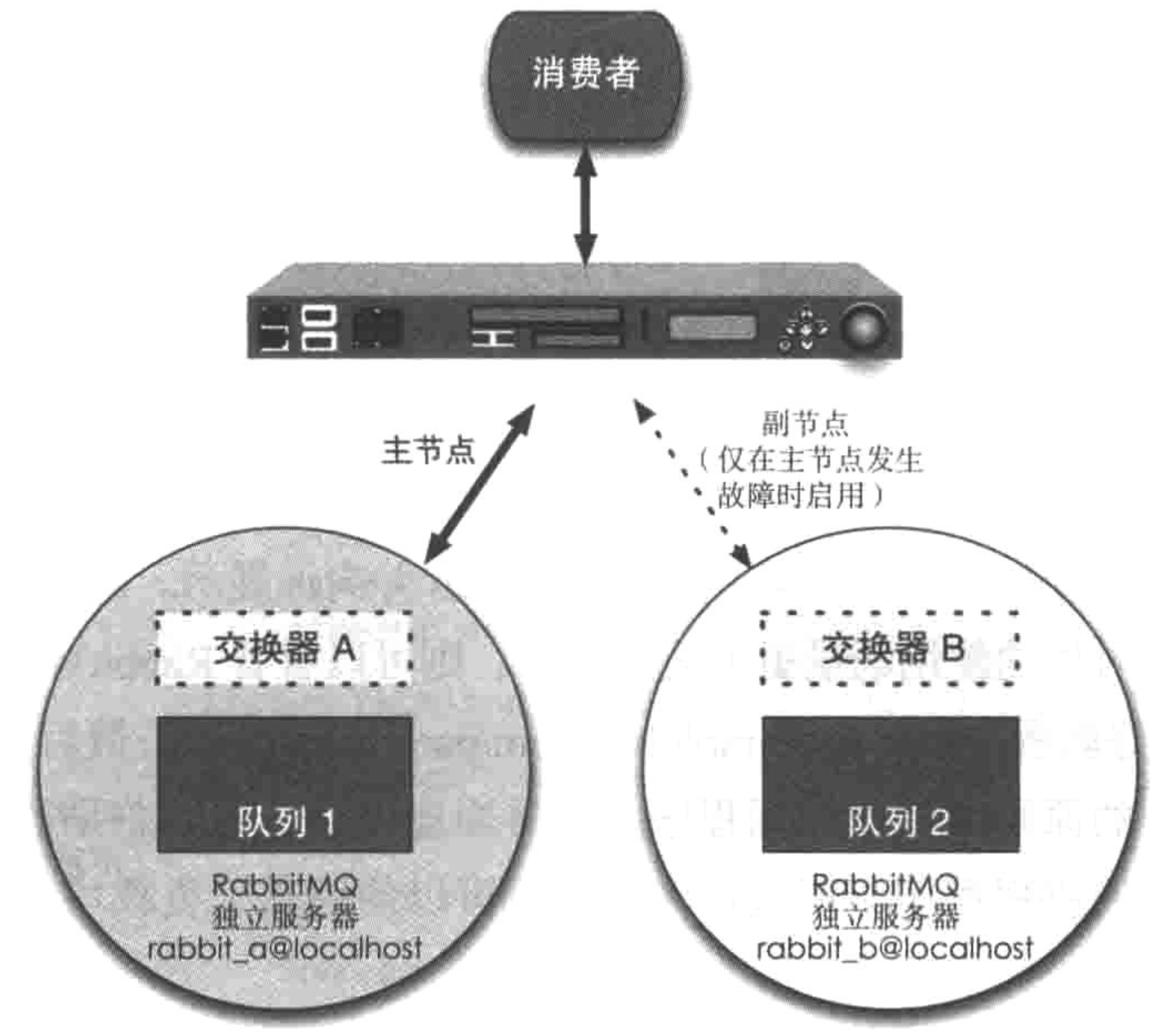

图 7.1 基于 warren 的负载均衡器

1 Pacemaker是Linux下用于在主备节点之间处理IP故障转移的一系列集群工具，并在故障发生时，在备用节点上自动启动受保护的应用。Pacemaker可以从http://www.clusterlabs.org/下载。

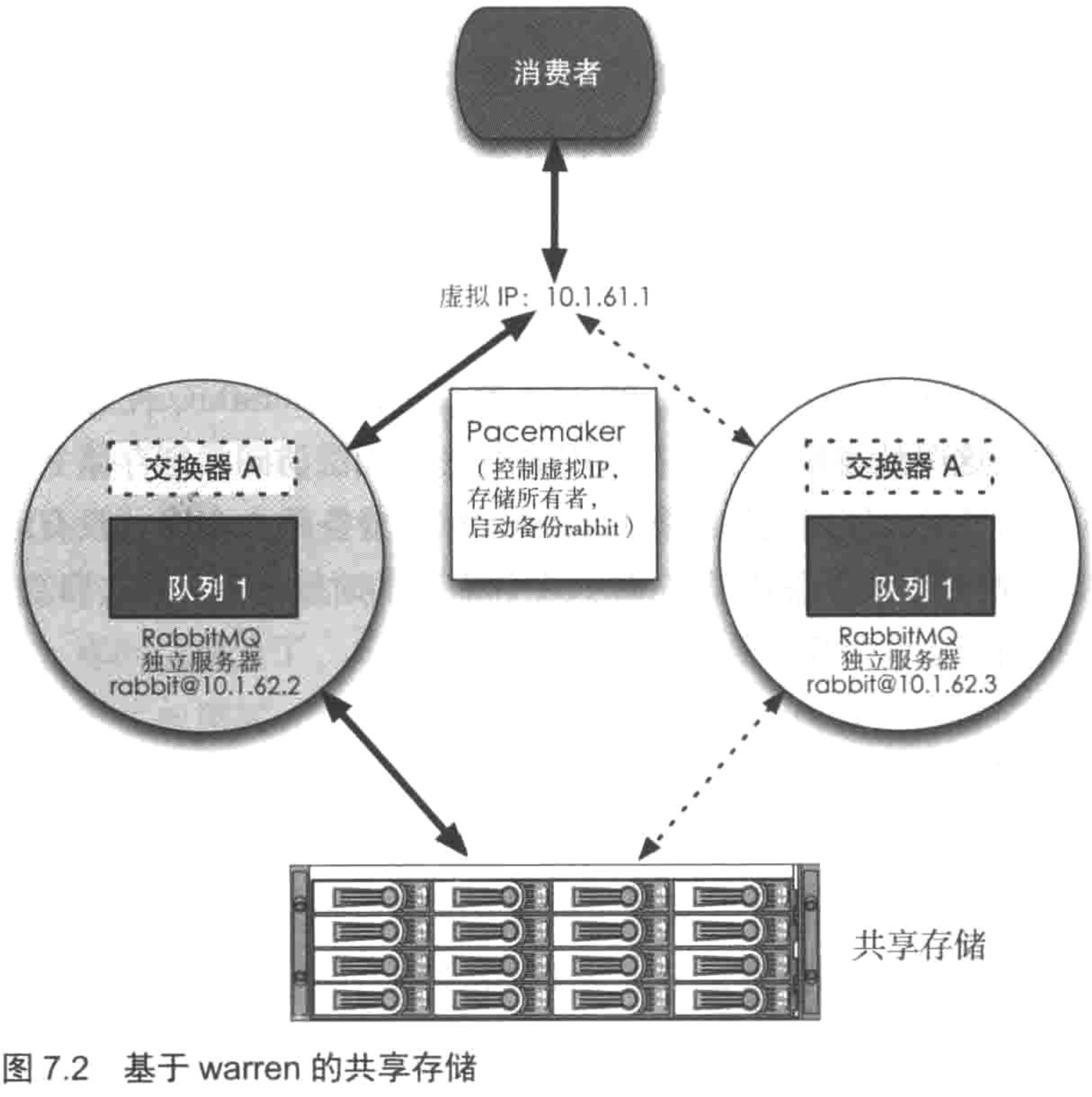

图 7.2 基于 warren 的共享存储

由于崩溃会在主备机之间复制的事实，并且在使用共享存储方法时无法在两个节点上都运行 RabbitMQ，因此我们选择基于负载均衡器的 warren 并向你展示。但是，如果共享存储的 warren 听起来对你的使用场景更为适用的话，则可以看看 RabbitMQ 网站上的这份关于如何设置的好教程：http://www.rabbitmq.com/pacemaker.html。最后，如果你使用“不做任何假设”的原则来构建应用程序，并且当连接服务器时总是重新搭建通信结构的话，那么你的应用程序不用做任何更改就可以使用基于负载均衡器的 warren 了。

7.2 设定负载均衡器——基于主/从的集群

事实上，设置一个基于负载均衡器的 warren 是非常简单的，运用到的都是到目前为止你所掌握的那些概念。首先，在你的开发系统上启动两个 RabbitMQ 节点，并将它们分别命名为 `rabbit_a` 和 `rabbit_b`，以避免和已经运行的本地集群节点相冲突（从 Rabbit 的安装目录运行）：

```
$ RABBITMQ_NODE_PORT=5675 RABBITMQ_NODENAME=rabbit_a \
./sbin/rabbitmq-server -detached
Activating RabbitMQ plugins ...
$ RABBITMQ_NODE_PORT=5676 RABBITMQ_NODENAME=rabbit_b \
./sbin/rabbitmq-server -detached
Activating RabbitMQ plugins ...
```

在启动主 / 备节点之后，现在需要设置一个新的 HAProxy 配置文件，将 rabbit_b 设置成备用服务器，只在 rabbit_a 发生故障时才启动。事实上，这份 HAProxy 的 warren 配置文件（如下列清单所示）看起来非常像集群配置。

清单 7.1 基于 warren 的负载均衡器的 HAProxy 配置

```
# HAProxy Config for Local RabbitMQ SLB Warren

global                                                          <-- 日志选项
    log 127.0.0.1    local0 info
    maxconn 4096
    stats socket /tmp/haproxy_2.socket uid haproxy mode 770 level admin
    daemon

defaults                                                        <-- 负载均衡默认配置
    log    global
    mode    tcp
    option    tcplog
    option    dontlognull
    retries    3
    option redispatch
    maxconn    2000
    timeout connect   5s
    timeout client 120s
    timeout server 120s

listen rabbitmq_local_cluster 127.0.0.1:5680                    <-- ❶ 前端IP供消费者和生产者使用

    mode tcp                                                    <-- 负载均衡选项
    balance roundrobin

    server rabbit_a 127.0.0.1:5675 check inter 5000 rise 2 fall 3   <-- 活动节点
    server rabbit_b 127.0.0.1:5676 backup check inter 5000 rise 2 fall 3   <-- 备用节点 ❷

listen private_monitoring :8101                                 <-- ❸ 数据统计页面
    mode http
    option httplog
    stats enable
    stats uri /stats
    stats refresh 5s
```

这里我们修改了 HAProxy 本地 socket 配置❶和数据统计页面❸，以免和 Rabbit 集群 HAProxy 实例相冲突。而更有趣的更改之处在于添加了一个新的 HAProxy 配

置选项❷：backup。当把 backup 添加到备用服务器指令时，就告诉了 HAProxy 只在所有的非备用服务器不可用时，才使用该备用服务器。如果你通过 http://localhost:8101/stats 查看 warren 的 HAProxy 数据统计页面（见图 7.3），就会发现 rabbit_b 是亮蓝色的，而非像 rabbit_a 那样的可用服务器是亮绿色的。亮蓝意味着 rabbit_b 是可用的但是是一个备用服务器（不可用的备用服务器是红色的，就像其他后台服务器那样）。

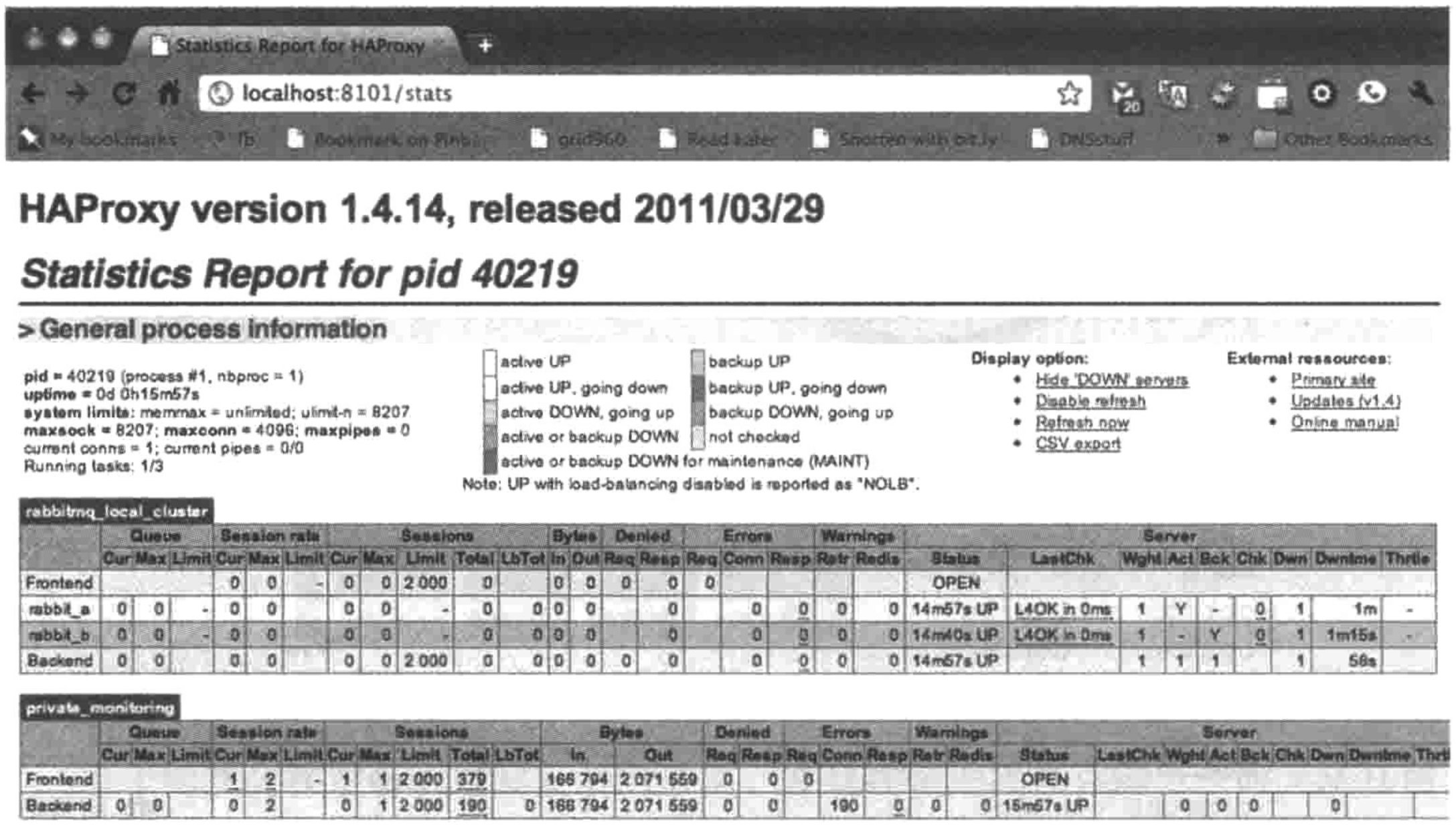

图 7.3 warren 的 HAProxy 数据统计页面

为了测试一下我们刚构造好的 warren，让我们从集群部分启动消费者，运行以下命令 python cluster_test_consumer.py localhost 5680：

```
$ python cluster_test_consumer.py localhost 5680
Ready for testing!
```

连接成功并准备行动。通过运行 python cluster_test_producer localhost 5680 来进行测试：

```
 [producer output]
Sent cluster test message.
 [consumer output]
Ready for testing!
Received: Cluster Test!/1301612486
```

看起来一切顺利。现在来测试故障转移。通过停止 rabbit_a 上的 Erlang 应用程序来产生故障。从 RabbitMQ 安装目录下运行 ./sbin/rabbitmqctl -n rabbit_a stop_app。现在记得你曾对 HAProxy 进行过设置，每隔 5 秒（inter 5000）对后台服务器进行健康检测，然后需要后台服务器失败三次（fall 3）之后才会认为其不可用了。所以需要等待 15 秒，消费者才会重连成功。一旦 HAProxy 故障转移到备用 RabbitMQ，你应该看到以下内容：

```
$ python cluster_test_consumer.py localhost 5680
Ready for testing!
Received: Cluster Test!/1301612486
Traceback (most recent call last):
  File "cluster_test_consumer.py", line 58, in <module>
    channel.start_consuming()
  File "/Library/Python/2.6/site-packages/pika-0.9.6-py2.6.egg/
  pika/adapters/blocking_connection.py", line 293, in start_consuming
    self.transport.connection.process_data_events()
  File "/Library/Python/2.6/site-packages/pika-0.9.6-py2.6.egg/
  pika/adapters/blocking_connection.py", line 87, in process_data_events
    raise AMQPConnectionError
AMQPConnectionError
...
Ready for testing!
```

如果你查看 HAProxy 数据统计页面的话（http://localhost:8101/stats），应该会看到在 Sessions 部分下的 Cur 头下有个数字 1。这说明你的消费者已经连接上了备用服务器。现在如果你再次使用 python cluster_test_producer localhost 5680 发布消息的话，消息仍然会到达消费者吗？

```
  [producer output]
Sent cluster test message.
  [consumer output]
...
Ready for testing!
Received: Cluster Test!/1301612545
```

是的，成功了！你拥有了全功能的主 / 备机 RabbitMQ warren，并基于负载均衡器来做故障转移。最重要的是可以确保当故障转移发生时，你无须担心 RabbitMQ 无法在备用节点上启动，因为它已经在运行了！由于 Rabbit 始终在主节点和备用节点上运行，因此你可以始终对它们进行监控；如果备机在派上用场之前就变为不可用的话，你在第一时间就能发现。这通过使用共享存储的 warren 是无法做到的。

在掌握了集群和 warren 之后，你就能处理故障并在数据中心之上进行扩展了。

但是当你需要在不同的数据中心的 Rabbit 间复制消息时，该怎么办？这时，我们就需要 Shovel 了。

7.3 远距离通信和复制

在只有一个数据中心的时候，RabbitMQ 集群对于提升消息通信性能来说是很棒的方案；但是当需要把消息从一个城市的 Rabbit 服务器路由到另一个城市的 Rabbit 上的时候，它就显得束手无策了。你可能会尝试使用集群来连接地理位置不同的数据中心，但是这会遇到一些麻烦。首先，你失去了对 RabbitMQ 选择集群节点放置队列的控制。所以即使你在芝加哥拥有两个集群节点并在洛杉矶拥有第三个节点，也仍然无法确保队列 A 在一个城市而同时队列 C 在另一个城市。第二点，Erlang 的 OTP 通信框架无法忍受网络延迟。所以在芝加哥和洛杉矶之间昂贵的 WAN 连接开销会导致巨大的问题以及集群各种各样奇怪的行为。事实上，如果 WAN 连接发生故障，RabbitMQ 没有办法应对网络划分。现在你也许会想："那该如何使用 RabbitMQ 来处理不同地理位置的基础架构呢？"答案是使用 Shovel。但在使用之前，你需要知道 Shovel 工作机制的背景知识。

7.3.1 给 Rabbit 装备 Shovel：Shovel 插件介绍

Shovel 是 RabbitMQ 的一个插件，可以使你能够定义 RabbitMQ 上的队列和另一个 RabbitMQ 上的交换器之间的复制关系。Shovel 最初是由 LShift（Rabbit Techologies 的母公司之一）设计的，现在由 RabbitMQ 核心开发团队来维护。像其他绝大多数 RabbitMQ 插件一样，Shovel 是 RabbitMQ 自有的 Erlang 应用程序，会在 Rabbit 启动时被加载。不同于大部分插件，Shovel 没有和 Rabbit 核心进行深度集成——当你在 Shovel 中定义两台服务器间复制关系的时候，你需要指定两台服务器的完整 URL，包括用户名和密码（例如，`amqp://guest:guest@localhost:5675/`）。从某些方面讲，可以将 Shovel 编写为独立的 Erlang 应用而不是一个 RabbitMQ 插件。但是凭借作为 RabbitMQ 的插件，你可以依赖每次 RabbitMQ 服务器启动时自动启动 Shovel 和自定义复制关系。

也许真实世界的示例可以更好地展示 Shovel 是如何工作的。农夫 Jacques 运营着 Avocados Supreme Limited，加州南部一个巨大的牛油果养殖公司。Jacques 遇到了一个问题：多年来，Avocados Supreme 运营着在加州的戈利塔（Goleta）市的单

一仓库。但是最近一段时间，该仓库经营负荷 80%，并且有时候库存会告罄，一些订单因此遭到了延误。为了解决这个问题，Jacques 沿着那条路在卡平特里亚（Carpinteria）开设了第二间仓库（见图 7.4）。卡平特里亚仓库的唯一目的是承载额外的库存，并在戈利塔仓库不堪重负时运送订单。

图 7.4 仓库地图拓扑

整个 Avocados Supreme 运作中最闪亮的一点是它们的订单处理系统。该系统使用 RabbitMQ 将网站和在戈利塔仓库中的订单处理连接起来。现在卡平特里亚仓库上线运作了，Pierre（Avocado Supreme 的技术架构师）正深陷困境，试图找出如何连接到新的仓库以接收戈利塔无法满足的订单。由于他无法使用 RabbitMQ 集群来桥接这两个仓库，因此 Pierre 打算更新网站将消息同时发布到戈利塔和卡平特里亚上的 RabbitMQ 服务器。但是 Pierre 的担心之处在于这会减慢网站上的订单接收，因为在其向客户确认订单之前，网站应用不得不发布消息到戈利塔（网站所处的地方）和卡平特里亚（一个更长的往返，见图 7.5）。更不用提 Pierre 现在不得不修改

Web 应用才能完成目标了。想当初他选择消息通信的理由之一就是，不用更改前端代码就能路由消息。正当 Pierre 哀叹之际，他发现了 Shovel。

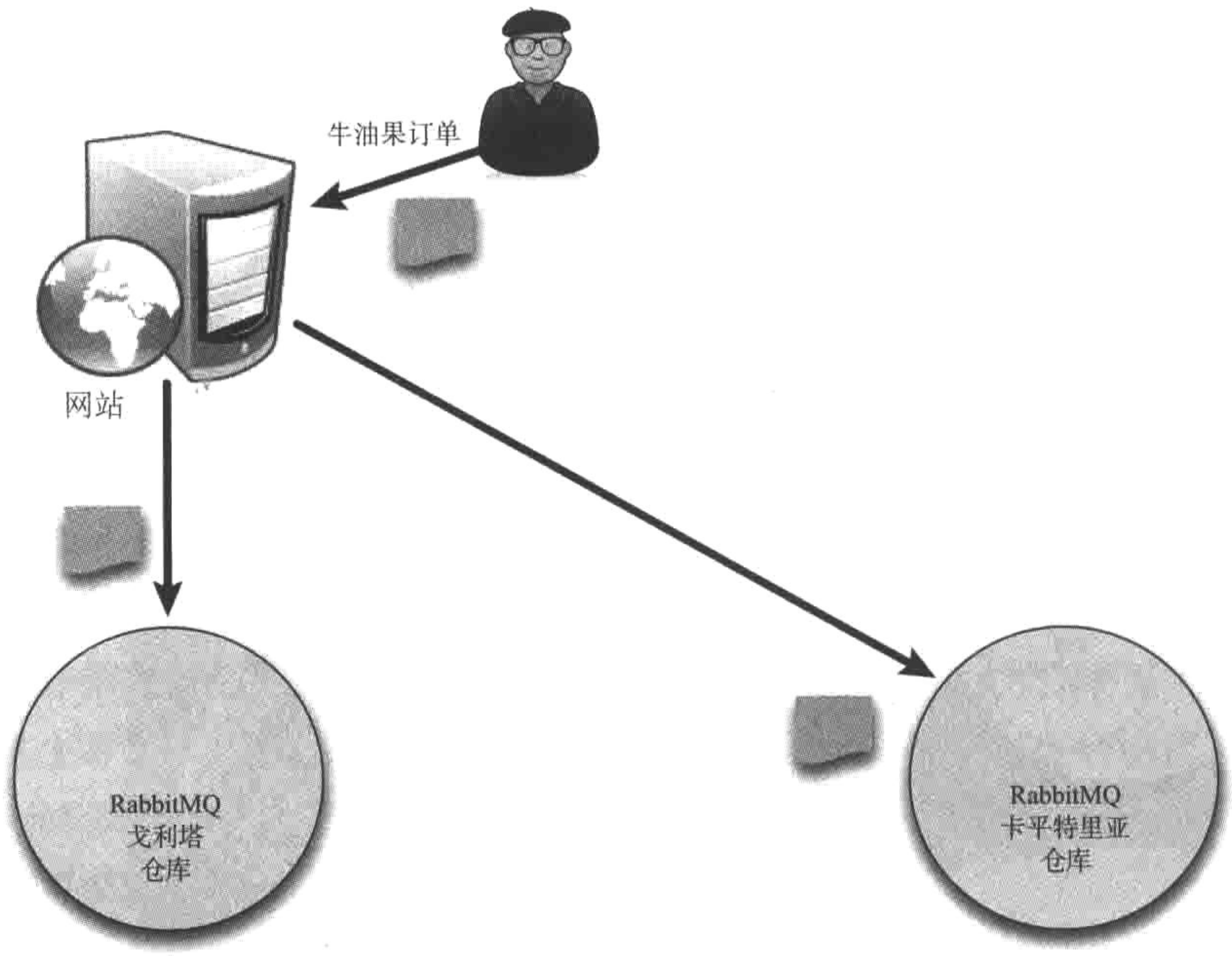

图 7.5 不使用 Shovel 的订单处理

在做了一些试验之后，Pierre 发现他可以使用 Shovel 在戈利塔创建一个新的队列，订阅到 `incoming_orders` 交换器上，用以接收网站发布的消息。然后他让 Shovel 消费这些消息并重新将消息通过 WAN 连接发布到卡平特里亚的 RabbitMQ 上的 `incoming_orders` 交换器（见图 7.6）。最棒的地方是网站不用放慢确认订单的脚步。它可以持续将消息发布到同一个 LAN 下的戈利塔 RabbitMQ 上，并给予牛油果爱好者们订单确认信息反馈。然后，Shovel 可以将那些订单异步复制到卡平特里亚，而不会让网站因为增加了网络延迟而有所觉察或者受到影响。这真是个完美的方案，Pierre 因为他的聪明才智享受了到蒙特雷的海滩度假。

因此，不管你需要在全国范围还是街道范围内，将消息在 RabbitMQ 服务器之间进行复制，Shovel 都是你值得信赖的解决方案。那么你如何安装 Shovel 并开启消息复制的魔法呢？

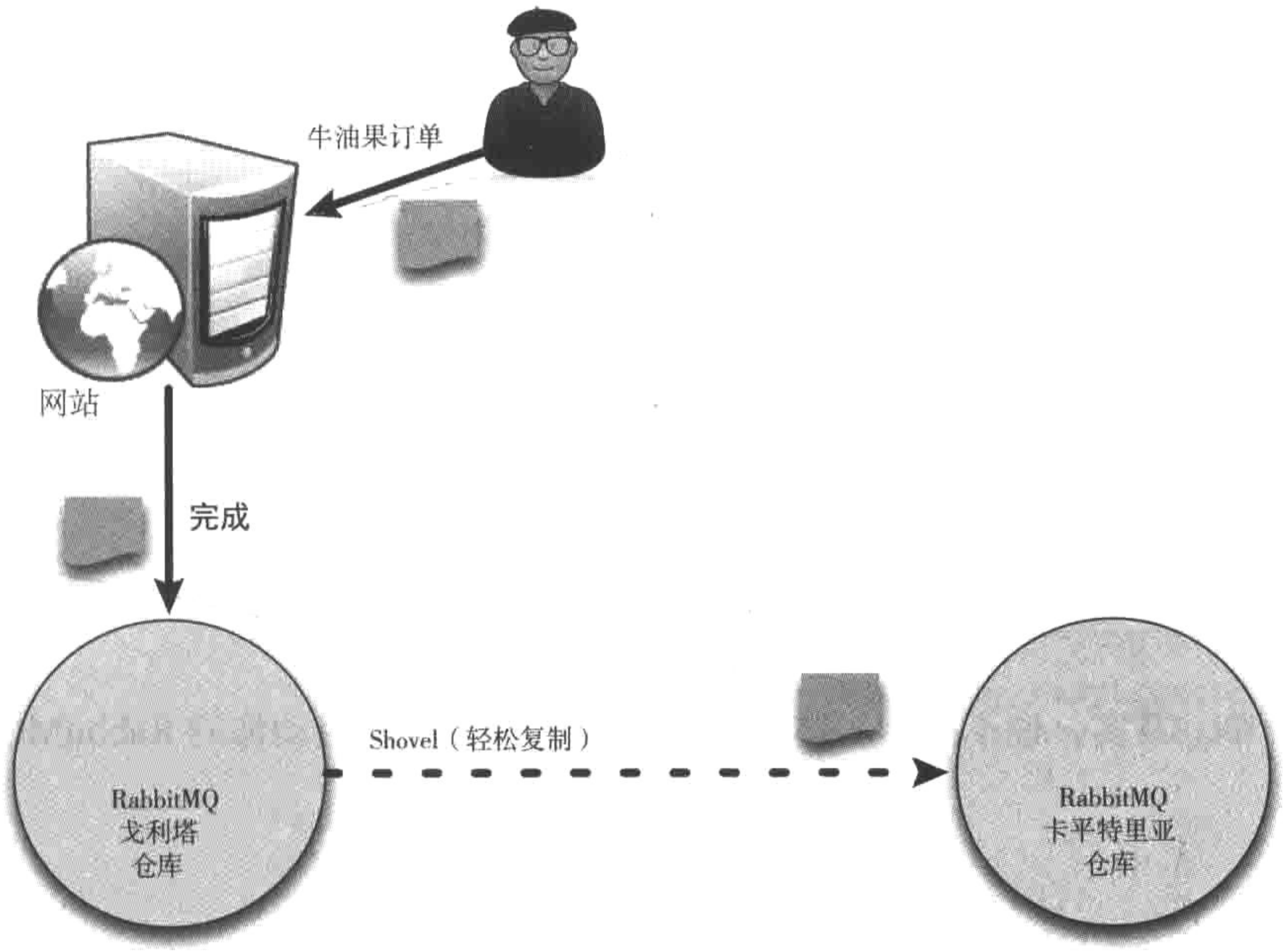

图 7.6 使用 Shovel 处理订单

7.3.2 安装 Shovel

Shovel 的安装是非常简单的。从 RabbitMQ 2.7.0 开始，`rabbitmq-shovel` 和 `amqp_client` 插件随 Rabbit 一起打包；你只需开启它们即可（在开启插件功能之前确保 `/etc/rabbitmq/` 存在）。在 RabbitMQ 工作目录下运行以下命令：

```
$ ./sbin/rabbitmq-plugins enable amqp_client
The following plugins have been enabled:
  amqp_client
Plugin configuration has changed. Restart RabbitMQ for
changes to take effect.
$ ./sbin/rabbitmq-plugins enable rabbitmq_shovel
The following plugins have been enabled:
  erlando
  rabbitmq_shovel
Plugin configuration has changed. Restart RabbitMQ for
changes to take effect.
```

如果你运行的是 RabbitMQ 2.7.0 以前的版本，则需要自行获取并安装插件。首先从 http://www.rabbitmq.com/plugins.html 下载 `rabbitmq-shovel` 和 `amqp_client` 插件，并将它们安装在 RabbitMQ 安装目录下的 `./plugins/` 目录：

```
$ cd ./plugins/
$ wget http://www.rabbitmq.com/.../v2.6.1/amqp_client-2.6.1.ez
Resolving www.rabbitmq.com... 208.91.1.36
Connecting to www.rabbitmq.com|208.91.1.36|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 154928 (151K) [application/andrew-inset]
Saving to: `amqp_client-2.6.1.ez'
100%[=================================>] 154,928 120K/s in 1.3s
$ wget http://www.rabbitmq.com/.../v2.6.1/rabbitmq-shovel-2.6.1.ez
Resolving www.rabbitmq.com... 208.91.1.36
Connecting to www.rabbitmq.com|208.91.1.36|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 42131 (41K) [application/andrew-inset]
Saving to: `rabbitmq-shovel-2.6.1.ez'
100%[=====================================>] 42,131 112K/s in 0.4s
```

就是这么简单。其实，也不尽然。你仍然需要配置 Shovel 并启动你的 RabbitMQ 代理服务器。

7.3.3 配置并运行 Shovel

所有 Shovel 的配置信息，包括从复制关系到重连设置，都包含在 `rabbitmq.config` 文件中。就像 RabbitMQ 配置文件中的其他部分一样，Shovel 的配置格式化名为 `rabbitmq_shovel` 的有效 Erlang 元组，并内嵌了配置指令。由于配置文件看起来十分复杂，因此让我们看看下列清单，它向你展示了 `rabbitmq.config` 中 Shovel 配置的面貌，然后我们再解读每个独立的部分。

清单 7.2 Shovel 配置文件 rabbitmq.config

```
[
    {mnesia, [{dump_log_write_threshold, 100}
              ]},
    {rabbit, [{vm_memory_high_watermark, 0.4}
              ]},
    {rabbitmq_shovel,
     [{shovels,
       [{avocado_order_shovel,
         [{sources, [{broker, "amqp://guest:guest@localhost:5675/"},
                     {declarations,
                        [{'queue.declare',
                                [{queue, <<"backup_orders">>},
                                 durable]},
                                {'exchange.declare',
                                   [{exchange, <<"incoming_orders">>},
                                    {type, <<"direct">>},
                                    durable]},
```

```
                    {'queue.bind',
                      [{exchange, <<"incoming_orders">>},
                       {queue, <<"backup_orders">>},
                       {routing_key, <<"warehouse">>}]}
                  ]}]},
{destinations, [{broker, "amqp://guest:guest@localhost:5676"},
          {declarations,
             [{'queue.declare',
               [{queue, <<"warehouse_carpinteria">>},
                durable]},
              {'exchange.declare',
               [{exchange, <<"incoming_orders">>},
                {type, <<"direct">>},
                durable]},
              {'queue.bind',
               [{exchange, <<"incoming_orders">>},
                {queue, <<"warehouse_carpinteria">>},
                {routing_key, <<"warehouse">>}]}
             ]}]},
 {queue, <<"backup_orders">>},
 {ack_mode, no_ack},
 {publish_properties, [{delivery_mode, 2}]},
 {publish_fields, [{exchange, <<"incoming_orders">>},
                   {routing_key, <<"warehouse">>}]},
 {reconnect_delay, 5}
]}
          ]
        }]
      }
].
```

就在 rabbitmq_shovel 指令的下方，你看到的是 shovels 部分。这是一个 Shovel 定义列表，其中每个 Shovel 定义了两台 RabbitMQ 服务器之间的复制关系。在这个例子中，你只有一个 avocado_order_shovel 的定义。在该 Shovel 中，你定义了消息复制的来源 sources 和消息发往的去处 destinations。sources 和 destinations 两者都包含了同样类型的配置指令：

- broker 或 brokers——URL，定义了用于 Shovel 两端的 RabbitMQ 服务器地址、用户名、密码和 vhost。如果 sources 或者 destinations 是 RabbitMQ 集群的话，那么就使用 brokers，并在其后用多个 URL 字符串且以 [] 包裹起来（举例来说，["amqp://server1...", "amqp://server2..."]）。这样的定义能够使 Shovel 在主节点故障时转移到另一个集群节点。
- declarations——AMQP 命令列表，声明了队列、交换器和绑定用于 Shovel 操作。

declarations是最难以理解的部分。它们内嵌在一个列表（数组）之中，每个成员都是一个定义了AMQP命令的Erlang元组，并伴随着另一个为该AMQP命令提供参数的元组：

```
{declarations,
   [{'queue.declare',
           [{queue, <<"backup_orders">>},
            durable]},
    {'exchange.declare',
           [{exchange, <<"incoming_orders">>},
            {type, <<"direct">>},
            durable]},
    {'queue.bind',
        [{exchange, <<"incoming_orders">>},
         {queue, <<"backup_orders">>},
         {routing_key, <<"warehouse">>}]}
]}]},
```

举例来说，该声明部分指示Shovel声明了一个叫作backup_orders的队列和一个叫作incoming_orders的交换器（记住我们的规则，不要对通信结构是否就绪做任何假设）。然后你告诉Shovel使用路由键warehouse将backup_orders绑定到incoming_orders上。你可能注意到了那些指令中一些有意思的地方。首先，所有字符串都不是简单地打上了引号；它们同时被双尖括号包围着：<<"backup_orders">>。这里的尖括号告诉Erlang不要将信息视为字符串，而是把它们视为特殊的数据类型*binary*。你不需要理解二进制是什么，只要知道如果你忘记添加尖括号的话，Shovel一启动就会崩溃。另一个有趣的事是durable参数没有像其他参数那样包裹在大括号内。这是因为durable不接收赋值。它要么存在,要么不存在。只有在参数需要赋值的时候才需要加上大括号。此外，你应该对AMQP命令和其参数很熟悉，而且你能指定自己所偏好的编程语言中拥有的任何参数。举个例子（虽然你不想这样做），你可以把backup_orders队列定义为auto_delete。auto_delete像durable一样，不接受赋值，就这样把它添加在queue.declare参数列表的durable之后（以逗号分隔）而不需要大括号。

在定义了sources和destinations之后，你也需要定义一些Shovel的通用设置。所有这些设置接收赋值，所以它们需要包含在大括号内。你可以定义的设置如下：

- queue——源服务器上的队列名称。Shovel会监听该队列以复制消息。

- ack_mode——在完成投递到目的服务器前，Shovel 是否应该确认接收源服务器的消息。
- prefetch_count——在任何时刻，Shovel 会在内部缓冲的消息条数。该内部缓冲区是一个源服务器和目的服务器之间的中途停留点。在 Shovel 发生故障时，它不受保护。
- publish_properties——当消息发往目的地时需要特别设置的属性列表。举例来说，{delivery_mode, 2} 将设置 delivery_mode 为可持久化的（2）。默认情况下，除非在 publish_properties 中对属性进行了覆盖，不然 Shovel 会在源消息发往目的地时复制源消息的属性。
- publish_fields——与 publish_properties 类似，定义了消息需要发往的目的服务器上的交换器，以及标记在消息上的 routing_key。如果 exchange 或者 routing_key 没有定义的话，Shovel 将会从原始消息上复制这些被忽略的设置。
- reconnect_delay——在连接断开之后，需要等待多少秒开始重连源服务器或者目的地服务器。

那么在这个例子中，你想要让 Shovel 从源服务器上的 backup_orders 队列消费消息，然后投递到目的服务器上的 incoming_orders 交换器，并设置消息的路由键为 warehouse。你不想对消息进行自动确认（举例来说，将 ack_mode 设置为 on_confirm 或者 on_publish），而想让消息以持久化的方式发布出去（delivery mode 2）。最后，你希望 Shovel 在连接断开后等待 5 秒之后再进行重连。将所有这些放在一块儿，那么配置文件看起来就像如下所示：

```
{queue, <<"backup_orders">>},
{ack_mode, no_ack},
{publish_properties, [{delivery_mode, 2}]},
{publish_fields, [{exchange, <<"incoming_orders">>},
                  {routing_key, <<"warehouse">>}]},
{reconnect_delay, 5}
```

图 7.7 展示了 Shovel 定义的拓扑图。你从 Rabbit 服务器 localhost:5675 上的 backup_orders 接收订单，并将它们发送到 Rabbit 服务器 localhost:5676 上的 incoming_orders 交换器上。

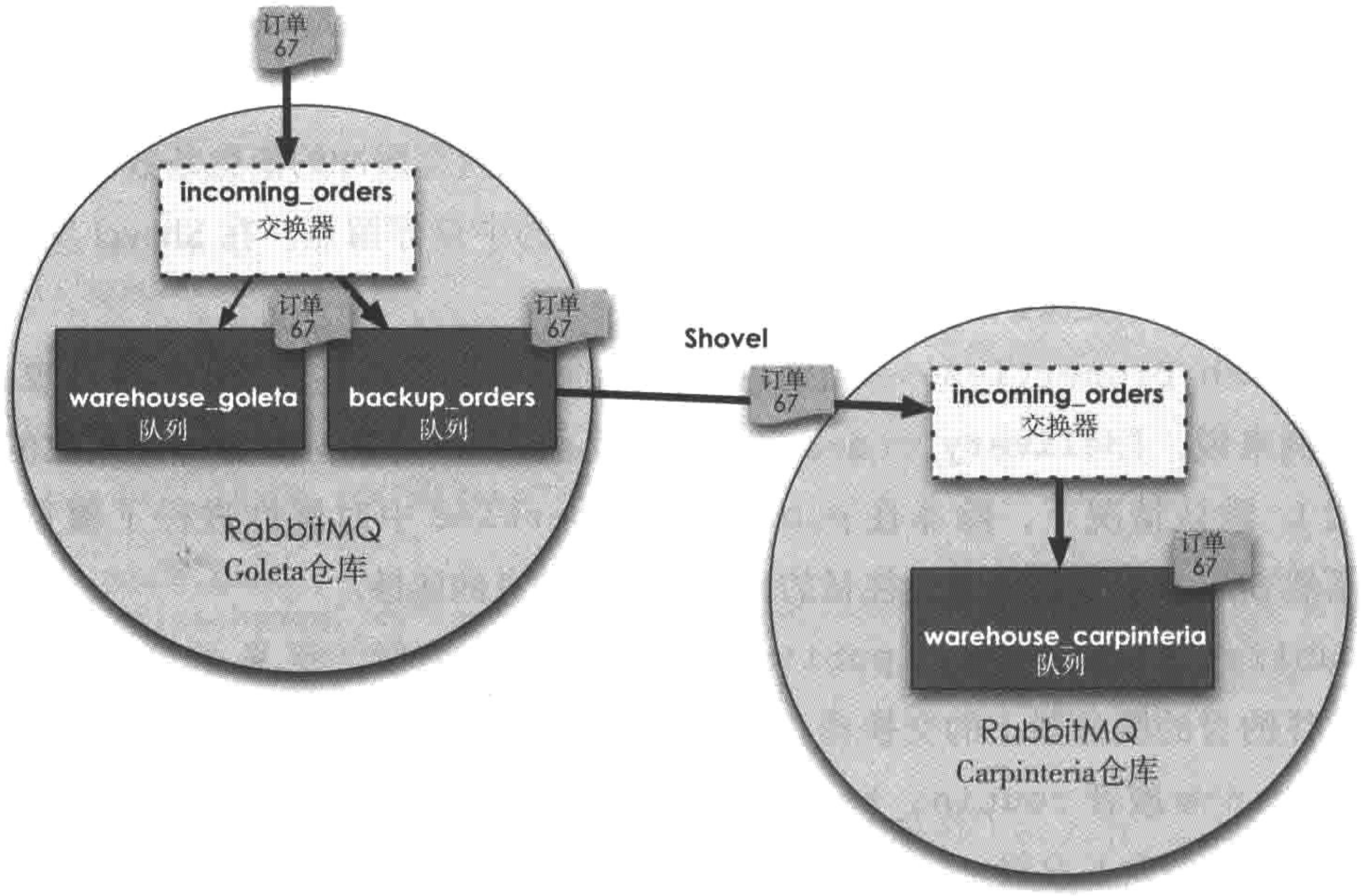

图 7.7 Shovel 订单处理拓扑

现在 Shovel 的配置算是完整了，是时候启动源服务器和目的服务器，看看其是如何工作的了。为了进行测试，你在开发系统上启动两个独立运行的 RabbitMQ 节点。由于源服务器和目的服务器都使用相同的 `rabbitmq.config` 文件，因此你最终将运行两个完全相同的 Shovel：一个在源服务器上，另一个在目的服务器上。通常情况下，你只想要一个 Shovel，但是既然你在本地运行两个节点，那么使用同一份配置文件来做测试会更方便。如果你仍然运行着 warren 那一章节的 `rabbit_a` 和 `rabbit_b` 那两台服务器的话，现在马上把它们停下来，因为你将在本例中重用它们。一旦停止之后，进入 RabbitMQ 安装目录并运行 `RABBITMQ_NODE_PORT=5675 RABBITMQ_NODENAME=rabbit_a ./sbin/rabbitmq-server` 来启动源节点（别使用 `-detached` 选项，这样你可以在配置文件中观察错误信息）。由于第一个终端已经被源服务器占用，因此开启一个新的终端并运行 `RABBITMQ_NODE_PORT=5676 RABBITMQ_NODENAME=rabbit_b ./sbin/rabbitmq-server` 来启动目的节点。在这两台服务器运行之后，你需要一个消费者和生产者以便测试。对集群的消费者和生产者稍加修改就可以处理牛油果订单了。首先，让我们看看消费者，如下列清单所示。

清单 7.3 Shovel 测试消费者

```
import sys, json, pika, time, traceback
def msg_rcvd(channel, method, header, body):
    message = json.loads(body)

    print "Received order %(ordernum)d for %(type)s." % message
    channel.basic_ack(delivery_tag=method.delivery_tag)

if __name__ == "__main__":
    AMQP_SERVER = sys.argv[1]
    AMQP_PORT = int(sys.argv[2])
    creds_broker = pika.PlainCredentials("guest", "guest")
    conn_params = pika.ConnectionParameters( AMQP_SERVER,
                                             port=AMQP_PORT,
                                             virtual_host="/",
                                             credentials=creds_broker)

    conn_broker = pika.BlockingConnection(conn_params)
    channel = conn_broker.channel()

    print "Ready for orders!"
    channel.basic_consume( msg_rcvd,
                           queue="warehouse_carpinteria",
                           no_ack=False,
                           consumer_tag="order_processor")
    channel.start_consuming()
```

打印订单并确认

服务器设置

建立服务器连接设置

发起到RabbitMQ的连接

开始处理订单

这和你在第 4 章中用来处理告警的那段消费者代码没太大区别。生产者（如下列清单所示）就更加简单了，仅仅更改了消息发送的牛油果订单格式。

清单 7.4 Shovel 测试生产者

```
import sys, json, pika, random
AMQP_HOST = sys.argv[1]
AMQP_PORT = int(sys.argv[2])
AVOCADO_TYPE = sys.argv[3]

creds_broker = pika.PlainCredentials("guest", "guest")
conn_params = pika.ConnectionParameters(AMQP_HOST,
                                        port=AMQP_PORT,
                                        virtual_host = "/",
                                        credentials = creds_broker)
conn_broker = pika.BlockingConnection(conn_params)
channel = conn_broker.channel()

msg = json.dumps({"ordernum": random.randrange(0, 100, 1),
                  "type" : AVOCADO_TYPE})
msg_props = pika.BasicProperties(content_type="application/json")
channel.basic_publish(body=msg, mandatory=True,
                      exchange="incoming_orders",
                      properties=msg_props,
                      routing_key="warehouse")
print "Sent avocado order message."
```

建立到服务器的连接

连接到RabbitMQ并发送消息

消费者接收两个命令行参数：目的服务器主机名和目的服务器端口号。生产者接收同样的参数，不过还多了第三个参数：牛油果类型（hass/ 哈斯、fuerte/ 富尔特等——任何单词都可以）。你将通过生产者向源节点发送牛油果订单，然后通过Shovel 的魔法，在另一个完全独立的目的服务器上的消费者接收并打印订单。让我们启动消费者并连接到目的节点吧：

```
$ python shovel_consumer.py localhost 5676
Ready for orders!
```

现在让我们向源节点发布一个牛油果订单：

```
$ python shovel_producer.py localhost 5675 hass
```

回到消费者终端，你应该看到如下所示的内容：

```
$ python shovel_consumer.py localhost 5676
Ready for orders!
Received order 66 for hass.
```

太棒了，你拥有了一个可以工作的 Shovel，使得发布到一台 RabbitMQ 服务器上的消息可以被完全无关的另一台服务器消费。虽然比配置集群更为单调，但是 Shovel 扩展了 RabbitMQ，允许你创建更为健壮的拓扑结构，将两台完全无关的 RabbitMQ 服务器和集群联合起来。在 RabbitMQ 原生支持远程复制之前，可以暂时先使用 Shovel 插件。

7.4 总结

将 Rabbit 打造得富有弹性是一个相当漫长的旅程，而在一开始的时候，你还任由服务器故障摆布从而导致 Rabbit 崩溃，并把消息通信架构“踢”下线。现在你不用再对供电中断提心吊胆了。如果你想要完全的高可用性，并且不接受任何消息丢失的情况下，就可以使用两台独立的 RabbitMQ 服务器来搭建 warren，并用负载均衡器使它们对应用程序来说像单个实体。如果由于某些原因，你需要这种可靠性扩展为将众多数据中心的 Rabbit 桥接起来，那么你知道该如何使用 Shovel 来提供复制功能。也许最重要的是，你知道如何使应用程序在面对单台 Rabbit 故障时快速恢复。不要对连接的服务器状态做任何假设，同时实现重连机制；满足以上两点，消费者

和生产者就能利用任何你学到的 RabbitMQ 冗余选项从节点故障中恢复，而不错过任何细枝末节。现在，你的应用程序和 Rabbit 足够健壮、富有弹性，并且已经为生产环境做好了准备！但是当健壮的分布式 RabbitMQ 基础架构对于管理员来说是一场痛苦的话，那还有什么用呢？为了回答这个问题，我们进入下一章，RabbitMQ Management 插件的美妙世界。

8 从Web端管理RabbitMQ

本章要点

- Management插件相对于rabbitmqctl脚本的优势
- 启用RabbitMQ Management 插件
- Management 插件功能
- 从Web控制台来管理用户、队列和交换器
- Management插件REST接口介绍

到目前为止，我们一直是通过 rabbitmqctl 脚本在命令行上来管理 RabbitMQ 的。在输入多次 rabbitmqctl 之后，你可能会梦想有一个图形界面来管理服务器，因为 MySQL 有 phpMyAdmin，CouchDB 有 Futon。对图形界面的需求导致了 RabbitMQ 社区创作了众多 RabbitMQ 的 Web 管理程序，它们之间差不多有相同的功能：展示队列状态，添加用户，创建 vhost，等等。值得庆幸的是，RabbitMQ 团队听取了来自社区的声音，开发了 RabbitMQ Management 插件。随着时间的推移，他们为了收集更多关于代理服务器使用状态而改进了服务器的 Erlang API。这些状态包括每秒消息发送数据、队列使用速率等。在本章中，你将会学习使用该插件，学会启用它并使用它的 Web 界面来管理用户、队列和交换器。进一步，你将学习新的 REST API，你可以通过新的 rabbitmqadmin 脚本轻松访问。

8.1 超越rabbitmqctl：RabbitMQ Management插件

重要的事情先来——到目前为止我们尚未讨论过插件。插件是一种扩展作者未能预见到的服务器的行为方式。RabbitMQ 的插件是由 Erlang 语言编写的，并且和服务器一同运行在同一个 Erlang VM 中。之后我们会有专门的一章来教你如何编写自己的 RabbitMQ 插件；因此在本章中我们将专注于如何启用并使用 Management 插件。让我们先看看为什么需要 Management 插件、它的功能是什么，以及如何在你的机器上启用并运行。

8.1.1 为何需要 Management 插件

比方说你很喜欢 `rabbitmqctl` 脚本（我们也一样）。如果你问为何需要使用该插件的话，以下是一些原因。

`rabbitmqctl` 脚本很棒，它让你可以做很多管理服务器的事情，但是它也有一些不足之处。首先，为了运行 `rabbitmqctl` 脚本，当前的 Linux 用户需要拥有访问用来启动服务器的 Erlang cookie 权限。由于服务器可能是以 `root` 或者 `rabbit` 用户身份来运行的，因此你需要对这些文件的访问权限。如果你是个人团队或者开发人员不多的话，这没什么问题。但如果是大型团队呢？为那个文件进行权限管理将变得一团糟。你会在团队成员中分享密码吗？故事到这里还没结束。当你拥有 `.erlang.cookie` 文件的访问权限时，这意味着你可以从 Erlang 控制台直接连接到 RabbitMQ 进程；也意味着你可以在服务器上执行破坏性的操作——不管是有意还是无意——没人可以阻止你。

除了安全问题之外，并不是所有的项目团队成员都是 CLI 的粉丝。我们工作过的几个项目中，甚至连产品负责人也想知道队列还留有多少后台通知。此外，有些时候你只想单击看看着色的信息，这比 `rabbitmqctl` 脚本生成的文本输出要容易理解多了。

8.1.2 Management 插件功能

Management 插件长什么样子呢？图 8.1 向你展示了不错的 Web 界面，你可以访问下列功能：

图 8.1 RabbitMQ Management 插件主界面

- 服务器数据统计概览——已投递的消息、服务器内存信息、Erlang 进程数等
- 导入 / 导出服务器配置
- 监控服务器连接
- 打开的信道列表
- 交换器列表、添加交换器
- 队列列表、添加队列
- 修改队列绑定
- 用户列表、添加用户
- vhost 列表、添加 vhost

现在让我们动手启用这个插件吧。

8.1.3 启用 Management 插件

使用最新版的 RabbitMQ（写作本书时为 2.7.0），插件的安装变得很简单。事实上，由于新版服务器发行包已经绑定了该插件，所以无须再进行安装了。唯一要做的事情是启用它们。如果进入 RabbitMQ 安装目录，运行以下命令你就能看到可用的插件了：

```
$ ls plugins/
README
amqp_client-2.7.0.ez
eldap-2.7.0-git.ez
erlando-2.7.0.ez
mochiweb-1.3-rmq2.7.0-git.ez
rabbitmq_auth_backend_ldap-2.7.0.ez
rabbitmq_auth_mechanism_ssl-2.7.0.ez
rabbitmq_consistent_hash_exchange-2.7.0.ez
rabbitmq_federation-2.7.0.ez
rabbitmq_jsonrpc-2.7.0.ez
rabbitmq_jsonrpc_channel-2.7.0.ez
rabbitmq_jsonrpc_channel_examples-2.7.0.ez
rabbitmq_management-2.7.0.ez
rabbitmq_management_agent-2.7.0.ez
rabbitmq_management_visualiser-2.7.0.ez
rabbitmq_mochiweb-2.7.0.ez
rabbitmq_shovel-2.7.0.ez
rabbitmq_shovel_management-2.7.0.ez
rabbitmq_stomp-2.7.0.ez
rabbitmq_tracing-2.7.0.ez
rfc4627_jsonrpc-2.7.0-git.ez
webmachine-1.7.0-rmq2.7.0-hg.ez
```

以 .ez 扩展名结尾的文件就是插件和支持的库了。举个例子，你想要启用的 Management 插件依赖于 AMQP Erlang 客户端 `amqp_client-2.7.0.ez` 和 webmachine 插件 `webmachine-1.7.0-rmq2.7.0-hg.ez` 都在插件列表之中。为了启用 Management 插件，你需要在代理服务器的 `sbin` 文件夹下运行以下命令：

```
$ ./rabbitmq-plugins enable rabbitmq_management
The following plugins have been enabled:
  mochiweb
  webmachine
  rabbitmq_mochiweb
  amqp_client
  rabbitmq_management_agent
  rabbitmq_management

Plugin configuration has changed. Restart RabbitMQ
for changes to take effect.
```

如你所见，`rabbitmq_management` 插件和与之相关的支持插件启用成功了[1]，但是你仍然需要重启代理服务器以使变更生效。

---

1 如果在运行./rabbitmq-plugins enable rabbitmq_management命令之后你得到了一个关于文件权限的错误的话，那么你很有可能需要修改/etc/rabbitmq/文件夹的权限设置。使用chown为运行rabbitmq进程的用户来更改该文件夹的权限。然后重新尝试启用该插件。

```
./rabbitmqctl stop
$ ./rabbitmq-server -detached
Activating RabbitMQ plugins ...
6 plugins activated:
* amqp_client-2.7.0
* mochiweb-1.3-rmq2.7.0-git
* rabbitmq_management-2.7.0
* rabbitmq_management_agent-2.7.0
* rabbitmq_mochiweb-2.7.0
* webmachine-1.7.0-rmq2.7.0-hg
```

如果一切正常的话，你可以用浏览器访问 http://localhost:55672/mgmt/，它会弹出一个认证窗口，需要你输入用户名和密码。此时你可以输入 `guest` 作为用户名和密码。一旦信息提交之后，你就会看到如图 8.1 所示的管理界面。如果你没有在 `localhost` 上运行服务器的话，那么就需要修改 URL 来适应你的环境。

根据上面这些简单的步骤，插件就已经设置好并且可以运行了。现在该学习如何使用它了。那么让我们这就进入下一节，通过敲击鼠标来管理 RabbitMQ 吧。

8.2 从Web控制台来管理RabbitMQ

让我们重新审视图 8.1。如你所见，最上面有个导航栏，在此你可以浏览 Connections、Exchanges 或者 Queues 等项目。然后，该界面展示服务器状态的概览。你可以看到所有的队列有多少条消息正准备投递出去，有多少条消息等待确认，以及消息总数是多少。这些信息对调试应用程序很有帮助，举个例子来说，等待确认的消息总数告诉你消费者消费消息的情况。如果该数字过高，那么可能就是消费者消费消息变慢的信号了。好处是你可以在前台页面就可以看到该信息，而不需要敲击 20 次才能看到。Web 控制台能做的远不止这些。在本节中，你将学会如何监控 Erlang VM，找出运行中的进程总数，同时也能看到如何将配置文件导出为 JSON 格式，以及如何将配置文件导回服务器。

8.2.1 监控 Erlang VM

如果把页面往下拉一点的话，你会看到一些关于运行着 Rabbit 的 Erlang 节点的有用信息。就如你在第 3 章看到的那样，你可以使用节点名称信息来远程连接到 RabbitMQ，并执行高级管理操作。另一个有趣的数字是 Erlang 进程总数：如果到

达上限的话，RabbitMQ 将停止工作。你可以通过修改 rabbitmq-server 初始脚本的 SERVER_ERL_ARGS 的 +P 选项来增加上限[2]。其他重要数值是安装的 RabbitMQ 和 Erlang 的版本。当你向 RabbitMQ 邮件列表发送 bug 报告时，你应该附上这些值。因为这样做能让那些诊断问题的人处理起来更容易，也更能帮到你。

在下一张表格里，你可以看到 RabbitMQ 当前正在监听哪个端口和主机。有多少次你抓破脑袋也想不出为什么连接不上服务器，而最后才发现问题在于错误的连接选项？在这里你可以查阅正确的选项，来避免类似的问题再次发生。

8.2.2 从 JSON 文件导入配置

当你浏览到 Overview 页面的结尾处时，会发现一个有趣的功能：你可以导出当前服务器的配置到一个 JSON 文件。你可以编辑之后再导回去。让我们看看当前机器上安装的服务器配置是怎样的。单击 Download Broker Configuration 按钮并把文件存储到硬盘。下列清单展示了格式化后的文件。根据你的设置，你下载的文件可能会略有不同。

清单 8.1 RabbitMQ JSON 配置

```
{
"rabbit_version":"2.3.1",

"users":
    [{"name":"guest",
      "password_hash":"6r578x5zS5/8oo1acUUiebYkRiU=",
      "administrator":true}],

"vhosts":[{"name":"/"}],
"permissions":[{"user":"guest", "vhost":"/", "configure":".*",
                "write":".*", "read":".*"}],

"queues":[
    {"name":"smart_proxy", "vhost":"/", "durable":true,
        "auto_delete":false, "arguments":{}},
    {"name":"control", "vhost":"/", "durable":true,
        "auto_delete":false, "arguments":{}}],

"exchanges":[
    {"name":"char_count_server", "vhost":"/", "type":"direct",
        "durable":true, "auto_delete":false, "internal":false,
        "arguments":{}},
```

2 该选项所在文件的位置会因为你安装RabbitMQ方式的不同而不同。如果你按照第1章的安装指引进行安装的话，那么rabbitmq-server就位于sbin目录中。

```
    {"name":"control", "vhost":"/", "type":"topic",
        "durable":true, "auto_delete":false, "internal":false,
        "arguments":{}},
    {"name":"char_count", "vhost":"/", "type":"direct",
        "durable":true, "auto_delete":false, "internal":false,
        "arguments":{}}],
"bindings":[]}
```

为了便于测试，我们尝试添加一个叫作 book 的新虚拟主机。将 vhost 行修改为下列代码片段：

```
"vhosts":[{"name":"/"}, {"name":"book"}],
```

保存更改之后，单击 Upload 按钮，选择修改过的 .json 文件进行上传。选择该文件之后，单击 Upload Broker Configuration 按钮。如果一切正常的话，你应该看到如图 8.2 所示的确认消息。通过使用这种简单的机制，更新服务器配置变得非常方便。同样，你可以导出配置文件，并加上版本信息。这样你可以记录不同时间点上的服务器配置。每次你对配置进行了修改之后，就可以保存旧的配置以防万一。

让我们继续探索 Management 插件提供的各种功能。在本例中，让我们看看你如何管理可以访问服务器的用户。

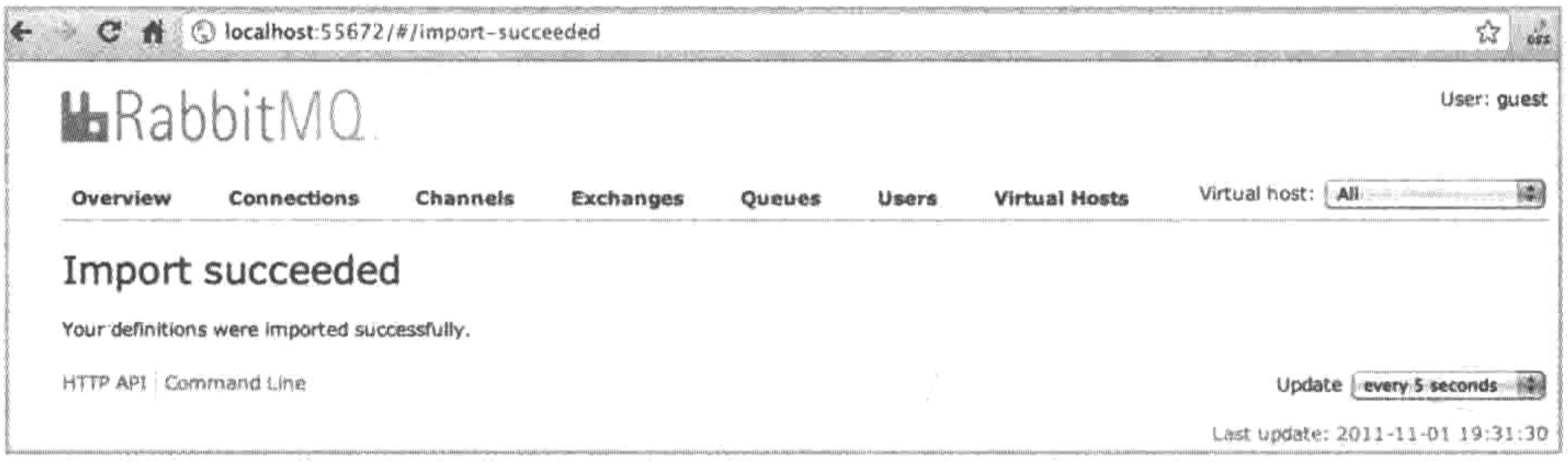

图 8.2 导入 JSON 配置

8.3 从Web控制台管理用户

下一步是学习如何从 Web 界面管理用户。在本书中，你一直使用默认的 guest 用户，对于学习目的来说这倒没什么；但是如果你想要在产品环境中运行 RabbitMQ 的话，就需要采取一些预防措施，创建你自己的用户和密码。所以在本节中，你将学习如何从 Web 控制台来完成简单的用户创建。用户管理并不只是用户创建；你也

需要赋予用户权限。8.3.2 节会介绍这一点。

8.3.1 创建用户

还记得当你第一次打开管理页面的时候，页面弹出表单让你输入用户名和密码，随后你输入了 `guest:guest` 吗？是时候换个账户了，因为你不会想以系统默认的用户设置来运行。单击上面导航菜单的 Users 连接，然后你应该看到如图 8.3 所示的对话框。

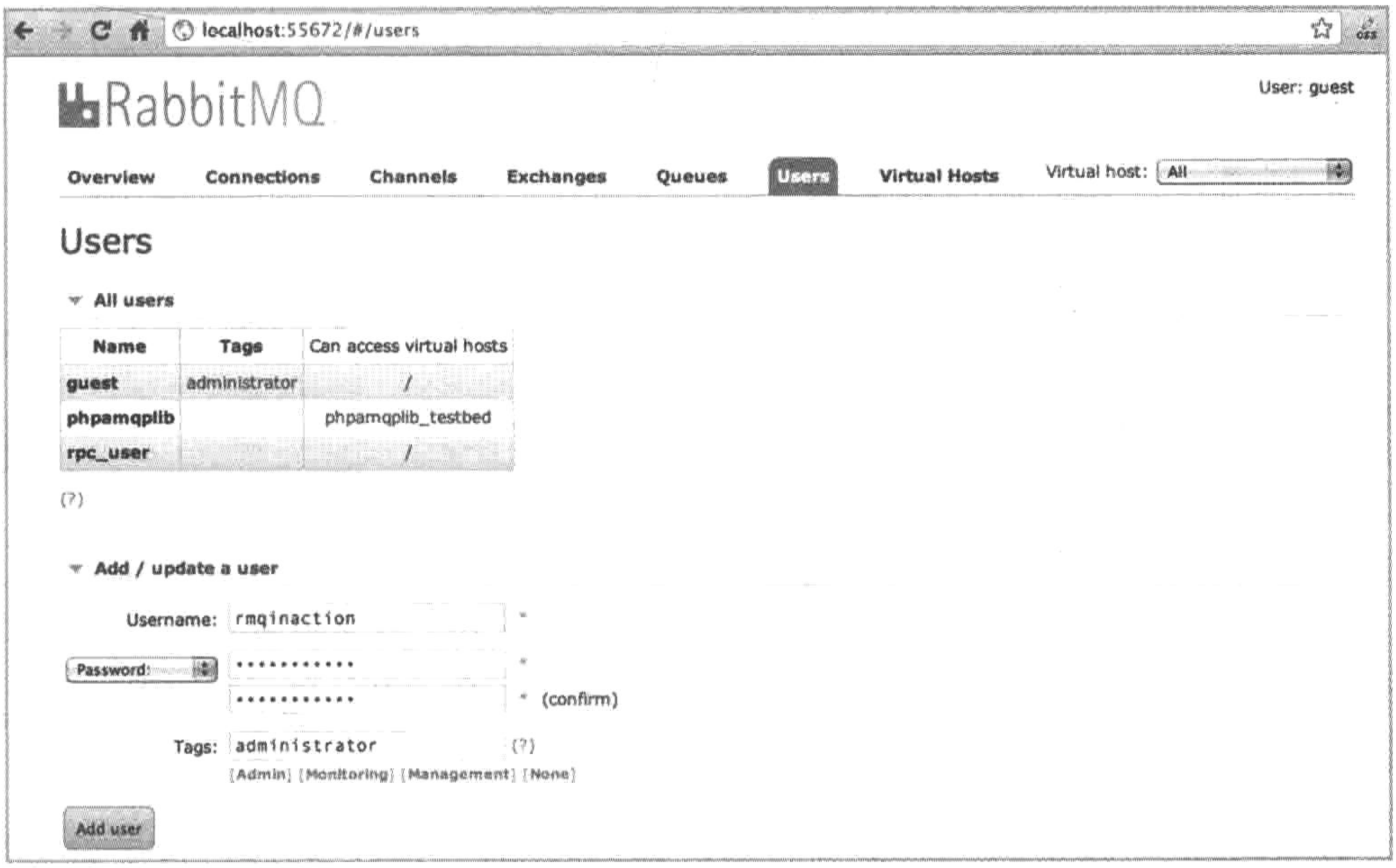

图 8.3 创建新用户

页面展示了系统的当前用户列表。列表下面是一张表单，可以用来添加新用户。如图 8.3 所示，创建了一个名为 `rmqinaction` 的用户并且密码也是 `rmqinaction`。最后，通过在 Tags 字段输入 `administrator` 来将用户设置成管理员。Management 插件支持用户角色的概念，用户角色决定了用户可以做什么。你的用户可以成为 `management` 角色，这意味着他们可以登录到这个界面。如果你将用户设置成 `monitoring` 角色，那么他们就能查看连接和节点相关的信息。`rmqinaction` 用户角色被设置成了 `administrator`，那么他就被授予了访问 Management 插件提供的所有功能。在图 8.4 中，你可以看到新用户的确认信息。

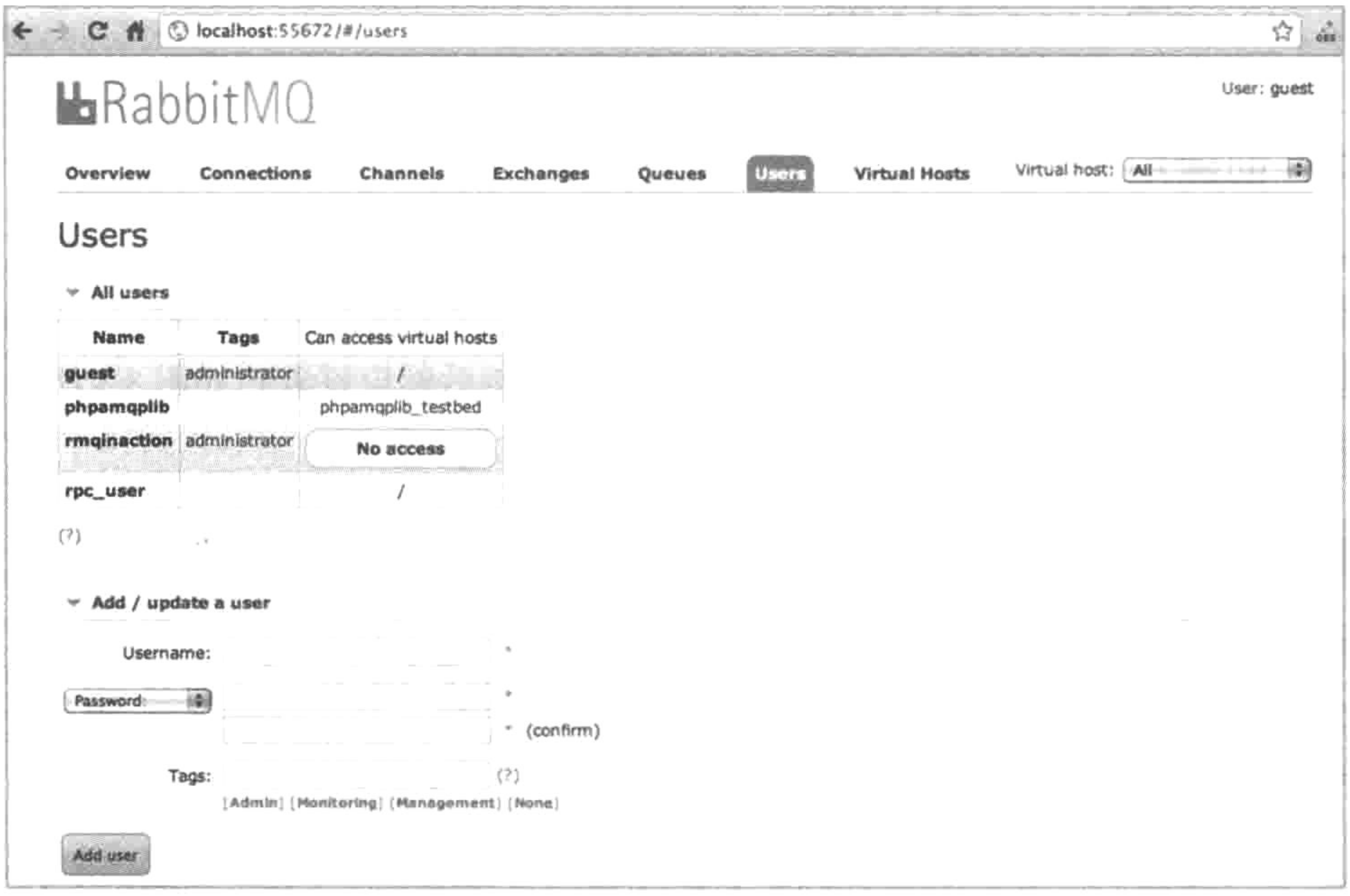

图 8.4 新用户确认

8.3.2 管理用户的权限

正如你从图 8.5 所看到的那样，你刚刚创建的那个用户没有权限访问任何虚拟主机。现在我们来为他授予权限。让我们授予他服务器配置和队列读写权限。单击用户名称，来到权限设置对话框，如图 8.5 所示。

使用图 8.5 中显示的 Management 插件提供的默认设置，因此只需单击 Set permission 按钮来保存变更。

作为用户管理的最后一处需要注意的地方，你可以从这张页面通过单击 Delete 按钮删除用户，如图 8.6 所示。

用户设置好了之后，下一步是学习如何管理队列和交换器。让我们仍然从浏览器上来完成这些操作。

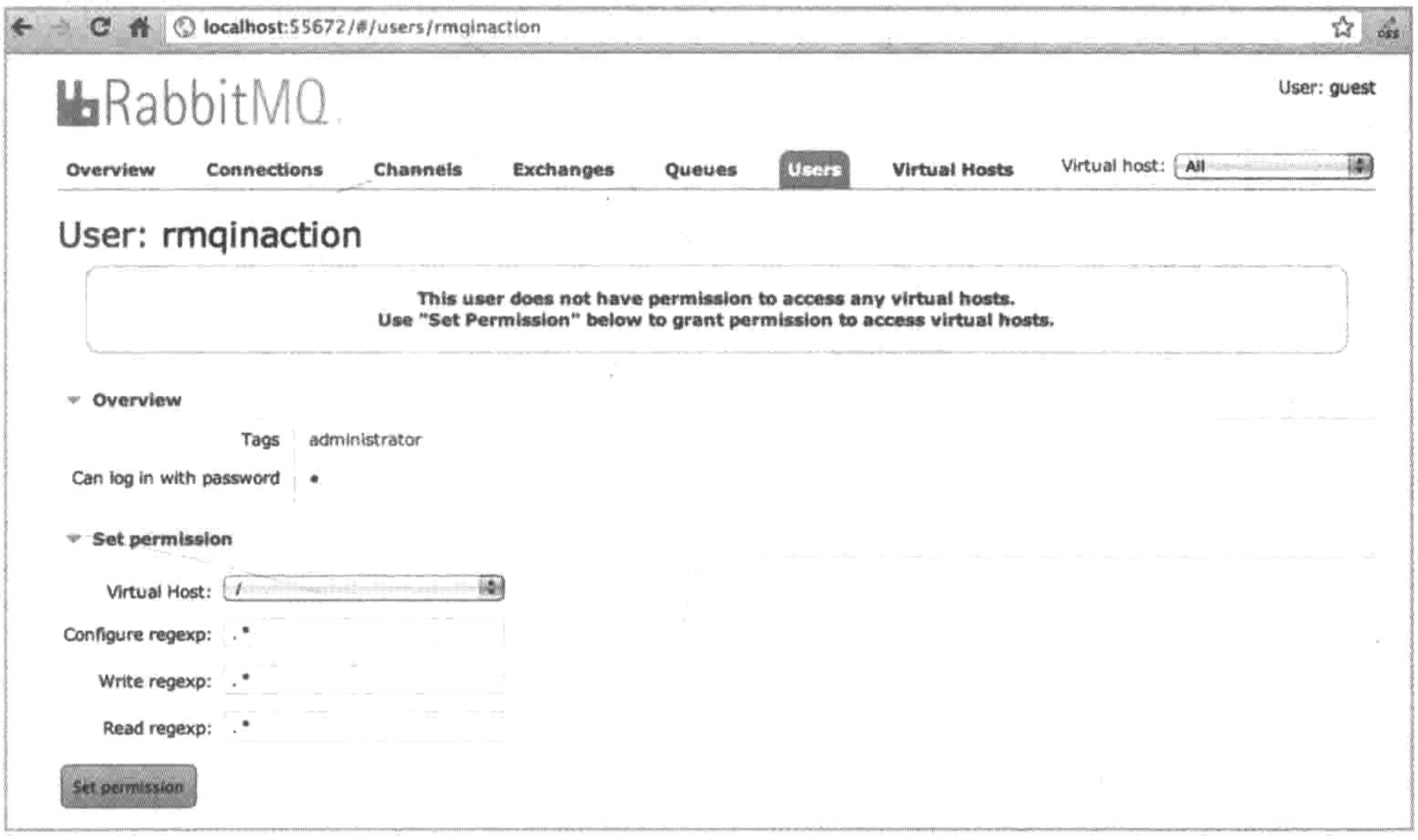

图 8.5 管理用户的权限

图 8.6 删除用户

8.4 从Web控制台管理交换器和队列

到目前为止你所做的一切都可以通过 `rabbitmqctl` 脚本来完成。但是，对于 `rabbitmqctl` 脚本来说，如果你想要创建交换器或者队列，那么你不得不求助于类库并编写一段脚本才行。而使用 Management 插件，你从浏览器就能创建交换器。

在导航菜单上单击 Exchanges 按钮。你会得到服务器上当前交换器的列表，如图 8.7 所示。

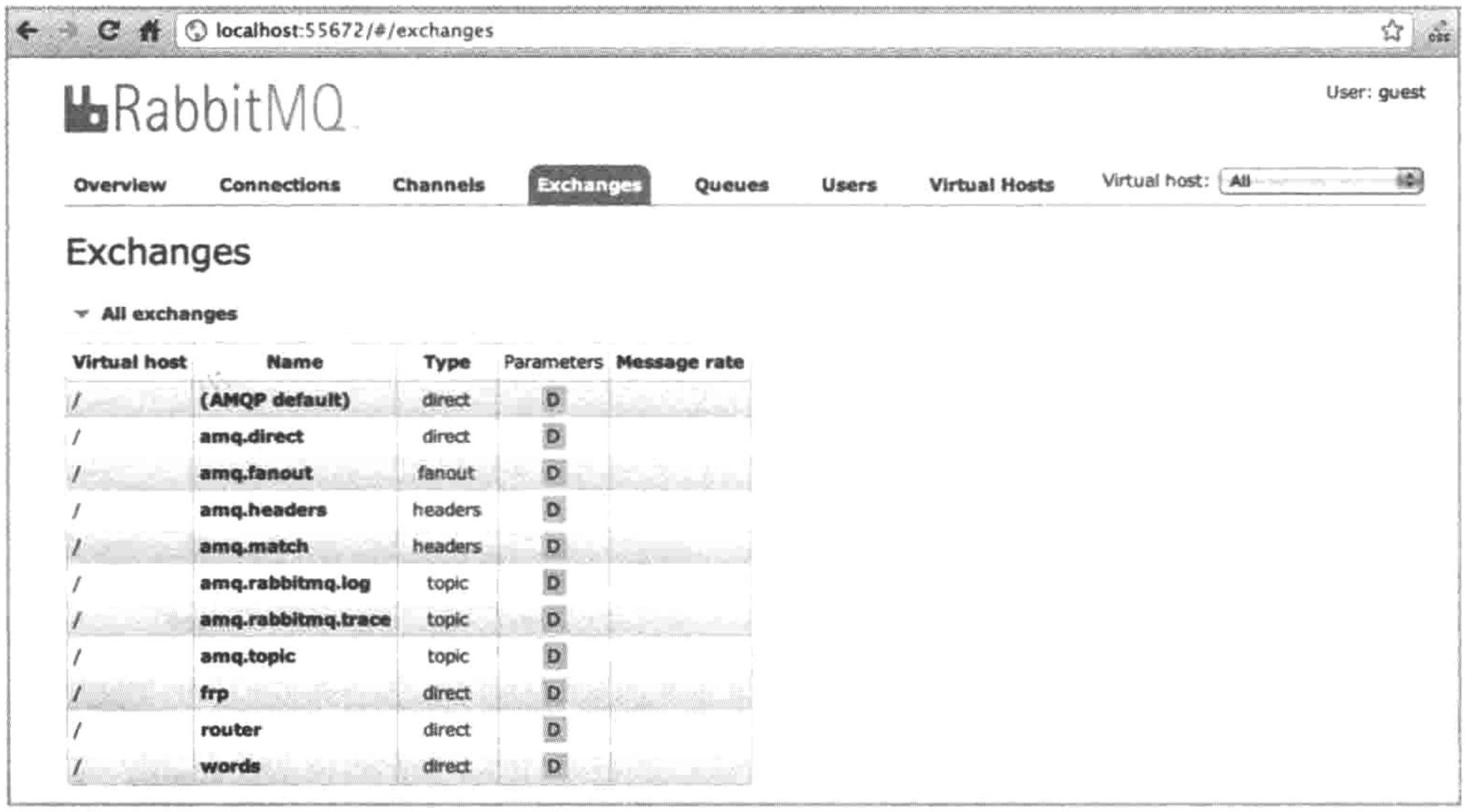

图 8.7 显示交换器列表

在交换器列表中，你可以获得以下信息：

- 交换器所存在的虚拟主机
- 交换器名称
- 交换器类型
- 参数列表，例如：D 代表持久化（durable）
- 消息速率（在往该交换器发送消息的情况下）

通过单击交换器名称，你可以看到更多详细信息，譬如交换器绑定。你也可以添加新的绑定，甚至彻底删除交换器，参见图 8.8 的示例。

最后，如果你回到交换器列表页面并往下翻的话，就会看到一张表单，允许你创建交换器，如图 8.9 所示。

让我们创建一个交换器，使用 test 作为名称，direct 作为交换器类型，然后其他选项使用表单的默认值。单击 Add exchange 按钮。瞧！在图 8.10 中，可以看到你新建的交换器了。只是单击了几次鼠标并敲了几下键盘就搞定了。

图 8.8 查看交换器详情

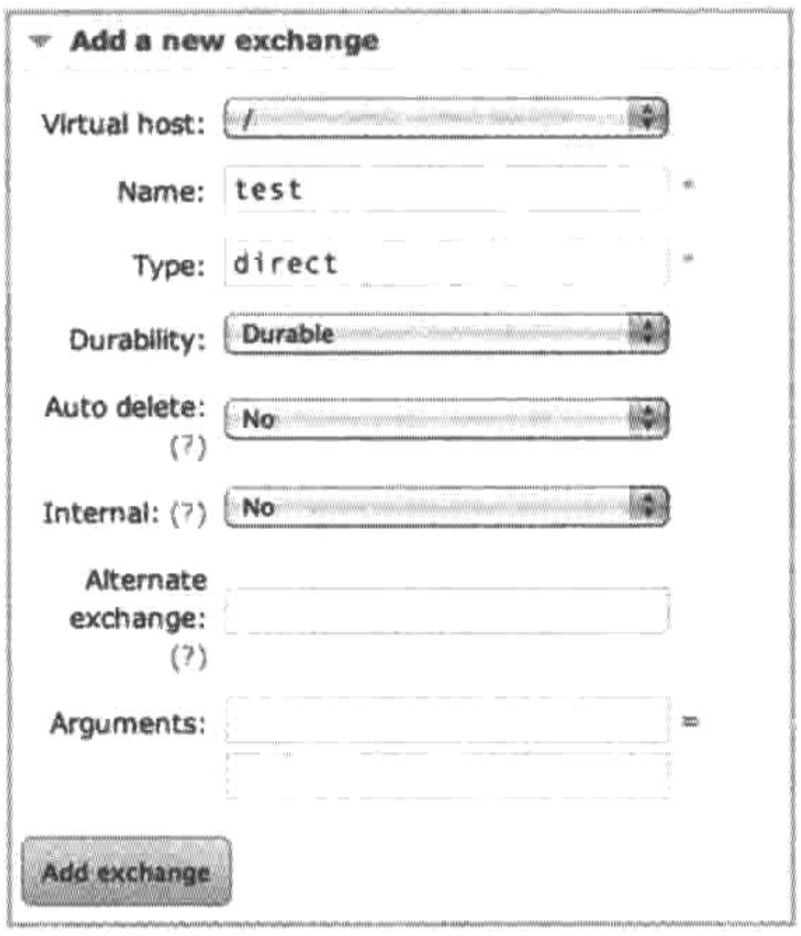

图 8.9 添加交换器

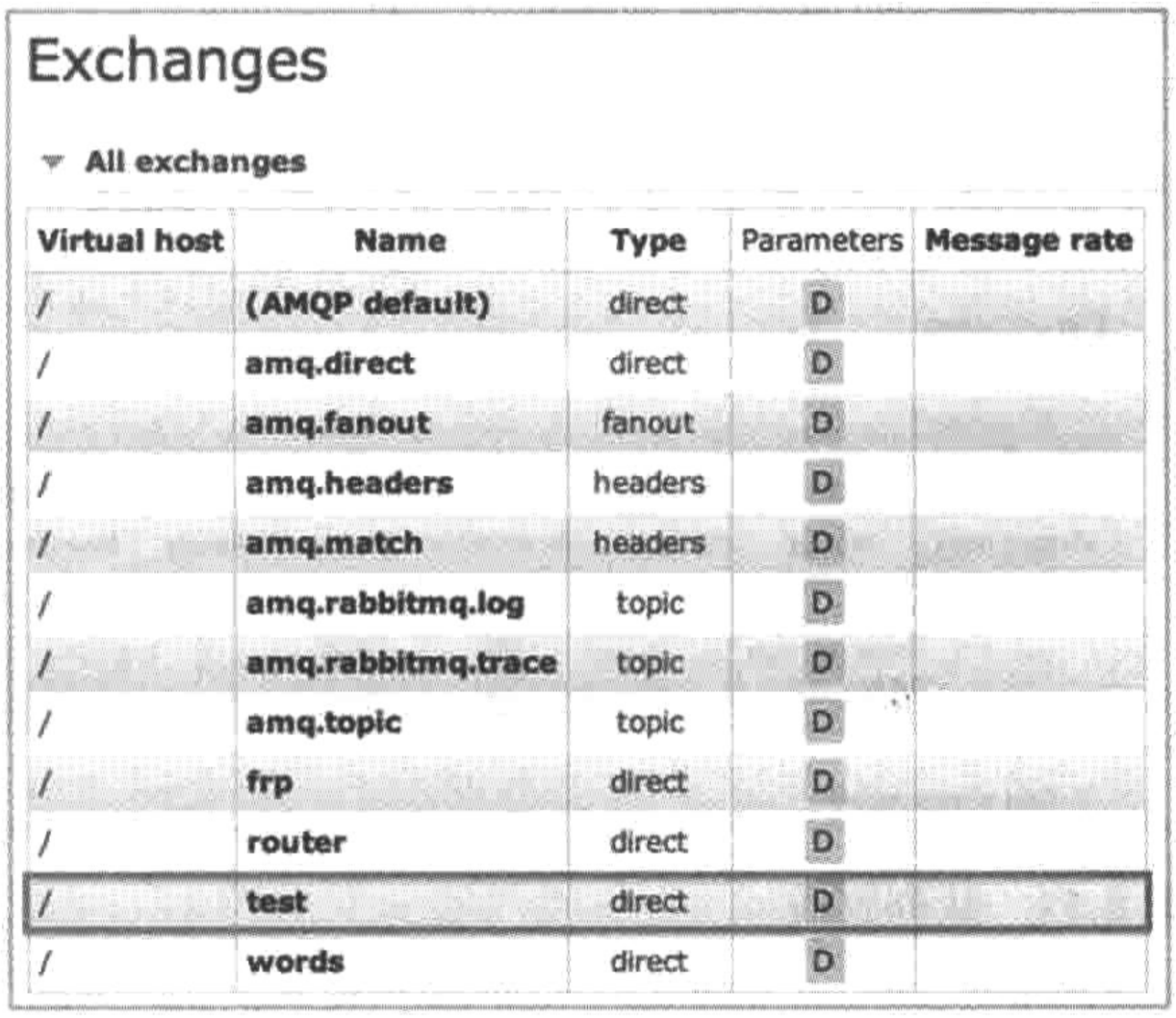

图 8.10 新的交换器

8.4.1 列出队列信息

在本章中，你已经见到了 Management 插件通过展现可视化通信结构来增加我们对 RabbitMQ 架构的理解。让我们比较一下 `rabbitmqctl` 和 Management Web 控

制台两者在队列清单上的异同。

进入 RabbitMQ 安装目录下的 sbin 文件夹，并运行以下命令：

```
$ ./rabbitmqctl list_queues -p '/' name messages_ready \
    messages_unacknowledged messages
Listing queues ...
char_count_server-queue    0    0    0
myQueueDE    0    0    0
smart_proxy    0    0    0
myQueueEN    0    0    0
control    0    0    0
...done.
```

然后回到浏览器上，并单击 Queues 链接。将这里显示的内容同你在图 8.11 所见到的比较一下。

甚至还有额外的惊喜，譬如显示队列是否是排他的、队列的状态以及队列的消息速率——后面的信息你只能通过 Management 插件获取。最后的这项功能看来已经说服你使用该插件了吧！

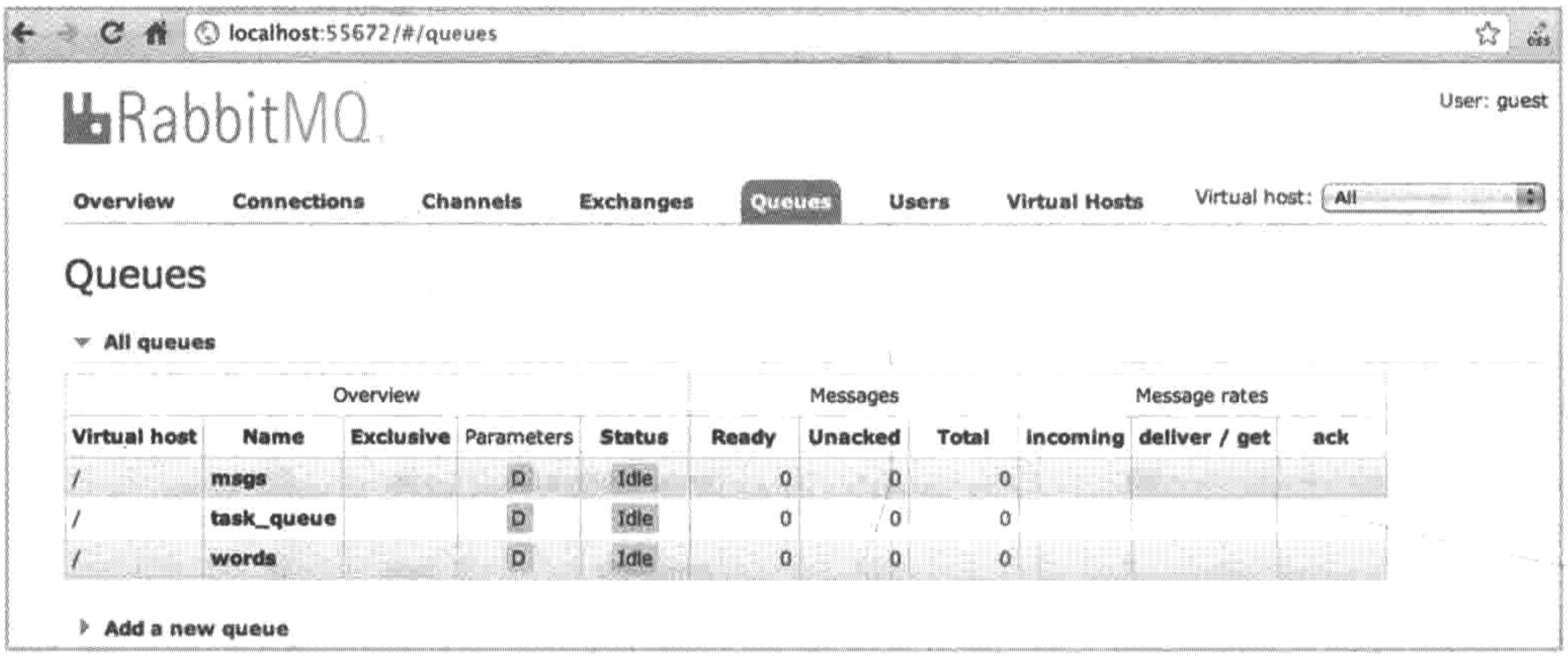

图 8.11 队列列表

8.4.2 创建队列

另一个你无法通过 rabbitmqctl 脚本做到的是创建队列。这当然可以通过 Web 控制台来完成。就如你创建交换器时那样，用 test 作为队列名，然后单击 Add queue 按钮。在图 8.12 中，你可以看到表单填写完后的样子。

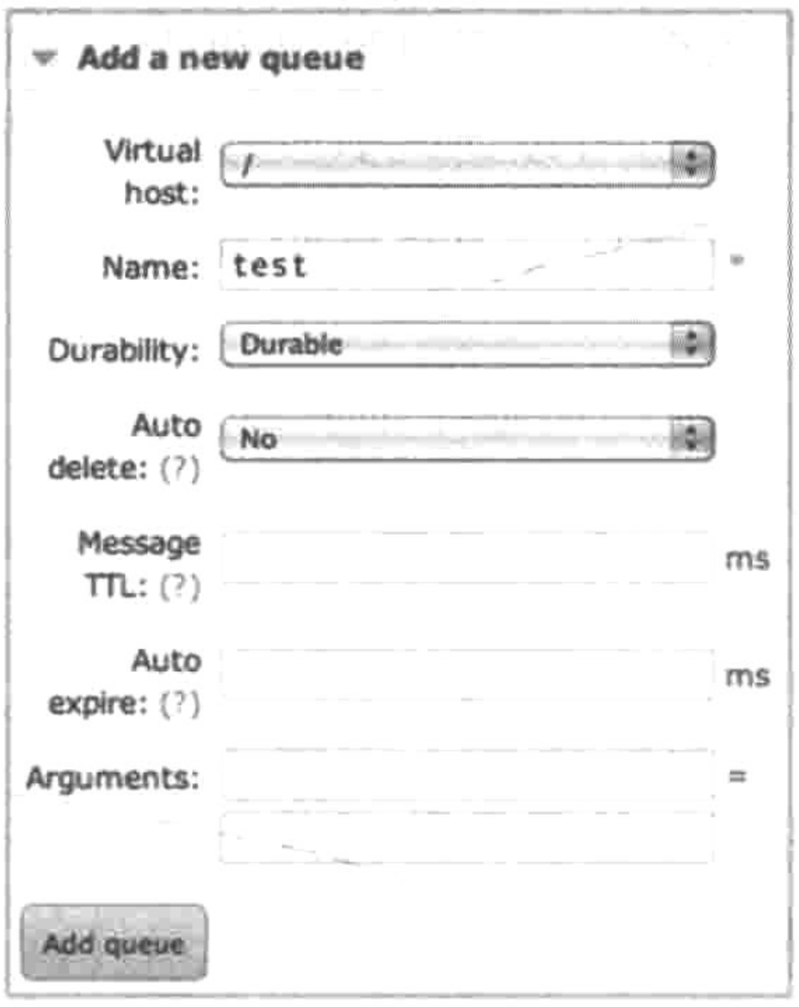

图 8.12 添加队列

新建队列会出现在列表当中。单击它可以查看队列的详细属性。有意思的是你可以通过浏览器直接删除或者清空它。这意味着如果服务器上的某个已不再使用的话，你只要单击 Delete 按钮，队列就删除了。

在图 8.13 中，你可以看到特定队列的状态详情，举例来说，有多少消息已经准备好投递给消费者，有多少消息等待确认。使用这个工具可以很方便地看到队列的内存使用情况，因此你可以使用该信息来监控系统的健康状况。如果使用 `rabbitmqctl` 脚本的话，则是无法完成这样的任务的。

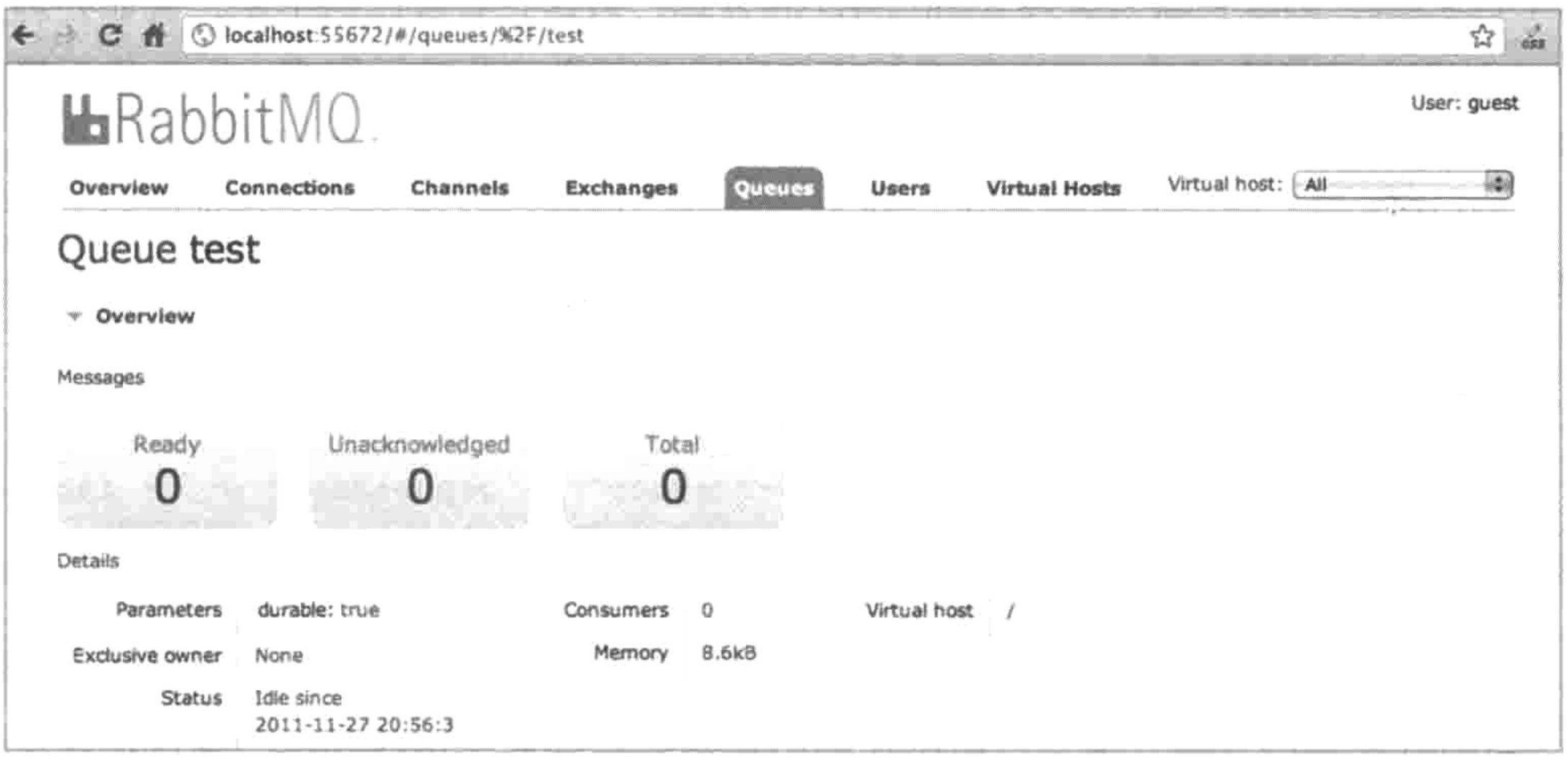

图 8.13 队列详情

Management 插件不仅仅是 Web 界面。为了真正成为系统管理员的好帮手，它需要提供一些命令行工具，帮助那些缺少视窗系统的机器也可以访问其功能，例如大多数 *nix 服务器。Management 插件也自带新的命令行接口，为管理过程增添了灵活性，将系统管理员从日常琐事中解放出来。

8.5 回到命令行

如果你可以将目前看到的一切全部自动化的话，那就非常了不得了——假设写一段脚本获得队列详情，譬如内存中等待确认的消息总数，并将其发布到诸如 Ganglia 或 Graphite 这样的图形工具上去。如果你想要通过 Management 插件的 Web 界面来完成上述工作的话，就不得不通过屏幕抓取的方式来做到这一点。这意味着如果新插件版本中的 Web 界面更改的话，你的脚本就失效了。一定有更好的实现方法。新的命令行工具 RabbitMQ Management CLI 让我们不用再受更多的折磨。在本节中，我们将讨论使用另一个命令行工具的原因。然后你将安装新的 `rabbitmqadmin` 脚本，并使用它来自动化任务，例如清空队列、创建交换器等。

8.5.1 为什么需要另一个 CLI

“等等……另一个 CLI 工具？但是在第 2 章你们已经教过我 `rabbitmqctl` 工具了啊。”我们理解你的这些疑惑。就如我们在本章开头提到的那样，虽然 `rabbitmqctl` 脚本是默认管理 RabbitMQ 的方式，但是它有自己的不足，譬如处理 Erlang cookie。除此以外，它也很难与其他编程语言和工具进行集成，因为你不得不解析它生成的自有格式输出。举个例子，如果你能获取队列列表的 JSON 格式并让 JSON 库来解析结果转换回 Python 哈希表的话，这样就好多了。正是这样的功能证明了学习 CLI API 是有价值的。让我们看看它是如何工作的。

用浏览器访问 http://localhost:55672/。你会看到下列三个选项：

- Management: Web UI
- Management: HTTP API
- Management: Command Line Tool

我们已经尝试过第一个了，所以让我们看看其他几个选项的功能。如果你单击 HTTP API，就会得到一个关于 REST 接口的文档页面。所有你能通过 Management

插件（Management Web UI）完成的功能，都可以通过使用 curl[3] 并调用 API 命令来完成。举个例子，如果想要列出服务器上的 vhost 的话，在终端执行下列代码：

```
$ curl -i -u guest:guest http://localhost:55672/api/vhosts
```

你会得到如下输出：

```
HTTP/1.1 200 OK
Server: MochiWeb/1.1 WebMachine/1.7 (participate in the frantic)
Date: Wed, 23 Mar 2011 20:07:22 GMT
Content-Type: application/json
Content-Length: 30
Cache-Control: no-cache

[{"name":"/"},{"name":"book"}]
```

你也可以在这里添加新的 vhost：

```
$ curl -i -u guest:guest -H "content-type:application/json" \
     -XPUT http://localhost:55672/api/vhosts/rmqinaction

HTTP/1.1 204 No Content
Server: MochiWeb/1.1 WebMachine/1.7 (participate in the frantic)
Date: Wed, 23 Mar 2011 20:12:28 GMT
Content-Type: application/json
Content-Length: 0
```

然后回到 vhost 列表，看看你刚刚添加的那个 vhost：

```
curl -i -u guest:guest \
http://localhost:55672/api/vhostsHTTP/1.1 200 OK
Server: MochiWeb/1.1 WebMachine/1.7 (participate in the frantic)
Date: Wed, 23 Mar 2011 20:12:57 GMT
Content-Type: application/json
Content-Length: 53
Cache-Control: no-cache
[{"name":"/"},{"name":"book"},{"name":"rmqinaction"}]
```

当然，如果不再需要的话，你可以删除它：

```
$ curl -i -u guest:guest -H "content-type:application/json" \
    -XDELETE http://localhost:55672/api/vhosts/rmqinaction
HTTP/1.1 204 No Content
Server: MochiWeb/1.1 WebMachine/1.7 (participate in the frantic)
Date: Wed, 23 Mar 2011 20:14:05 GMT
```

---

3 命令行工具curl用URL语法来转移数据。它支持众多格式，如HTTP、RTMP、IMAP等。如果你需要在本机安装的话，可以从网站http://curl.haxx.se/上进行下载。

```
Content-Type: application/json
Content-Length: 0
```

通过使用 REST API，你可以轻松自动化那些到目前为止只能通过图形化界面完成的任务。毫无疑问，系统管理员会非常满意这个工具。如果你想学习 REST API 提供的功能的话，就可以通过浏览器访问随 Management 插件一起发布的文档。http://localhost:55672/api 有你想要的内容。你可能注意到了响应头里有一些有趣的东西：应答是以 `application/json` 的方式发送的，这意味着需要用你拿手的编程语言的 JSON 库来解析结果（很可能你已经会了）。

8.5.2 CLI 管理：一种更简单的方式

之前提到的那个方法既方便又具灵活性，不过仍然有更简单的方法来管理服务器：那就是 command-line tool。该命令行工具是一段 Python 脚本，你可以从 RabbitMQ Management 插件安装目录直接下载并执行。该脚本相比基于 REST 的 API 的好处是，你不再需要手工编写请求了。`rabbitmqadmin` 会包装 REST API，让你使用干净的接口与其交互，所以取代下列命令（列出服务器上的队列），

```
$ curl -i -u guest:guest http://localhost:55672/api/queues
```

你可以这样做：

```
./rabbitmqadmin list queues
```

看起来不错吧？现在就让我们来安装它吧。

8.5.3 安装 rabbitmqadmin 脚本

安装 `rabbitmqadmin` 管理脚本简单得要命：唯一的需求是安装好 Python 即可。假设你已经安装好了 Python，接下去需要做的是从 Management 插件中获取该脚本，并设置成可执行文件。通过下列两个命令来完成：

```
$ wget http://localhost:55672/cli/rabbitmqadmin
    $ chmod +x rabbitmqadmin
```

这会将 `rabbitmqadmin` 脚本放置在当前目录下，可以通过终端轻松调用。通过使用该脚本你可以避免学习 `curl` 所有复杂的东西。让我们看看实际操作：

```
$ ./rabbitmqadmin -V "/" list exchanges
+-------+--------------------+---------+--------+---------+----------+
| vhost |        name        | type    | auto   | durable | internal |
|       |                    |         | delete |         |          |
+-------+--------------------+---------+--------+---------+----------+
| /     |                    | direct  | False  | True    | False    |
| /     | amq.direct         | direct  | False  | True    | False    |
| /     | amq.fanout         | fanout  | False  | True    | False    |
| /     | amq.headers        | headers | False  | True    | False    |
| /     | amq.match          | headers | False  | True    | False    |
| /     | amq.rabbitmq.log   | topic   | False  | True    | False    |
| /     | amq.topic          | topic   | False  | True    | False    |
| /     | test               | direct  | False  | True    | False    |
+-------+--------------------+---------+--------+---------+----------+
```

就像你看到的那样，输出的内容是格式化过的，展现在你眼前的是 "/" vhost 内交换器的相关信息。让我们拆解你刚才调用的命令：rabbitmqadmin 是可执行文件的名称，也就是那个刚刚下载的 Python 脚本；-V "/" 是用来确定想要使用的 vhost 的选项；最后，list exchanges 是你想要执行的命令。

8.5.4 清空队列、创建交换器等

假设出于某种原因，有个消费者表现得不太正常，无法消费消息，也无法向订阅的队列确认消息。你注意到了这个问题时，该队列已经填满了无意义的消息。你可以写一段简单的脚本使用 AMQP 来清空队列，或者可以简单地调用下列命令：

```
$ ./rabbitmqadmin purge queue name=test
queue purged
```

让我们回到创建新交换器的例子当中。让我们看看你在以用户名 guest 和密码 guest 认证之后，如何声明一个名为 cli_test 的交换器：

```
guest:
$ ./rabbitmqadmin -u guest -p guest declare exchange \
name=cli_test type=direct
exchange declared
```

另一个有趣的命令是关闭连接。假设消费者由于行为异常无法确认消息，因而扰乱了消息流。首先你可以获取连接列表，里面只有一个 connection name 属性：

```
$ ./rabbitmqadmin list connections name
+------------------+
|       name       |
+------------------+
| 127.0.0.1:64659  |
+------------------+
```

如果想要关闭该消费者的话，你可以通过调用close connection命令来完成：

```
$ ./rabbitmqadmin close connection name="127.0.0.1:64659"
connection closed
```

虽然没有详细描述每一个命令，但是这里展现了 rabbitmqadmin 脚本的基本使用方法。以此为基础，你可以开始执行其他任务。

8.6 总结

当提到系统管理的时候，一些人会选择命令行工具，而另一些则会偏向 GUI 程序。管理 RabbitMQ 服务器有一大堆选择。好在本章中我们展示的所有工具都是由 RabbitMQ 开发人员编写并维护的。根据个人爱好，你会更倾向于使用 Web UI。对于日常开发来说非常方便，你可以用视图查看服务器状态。Web UI 使得团队合作更加简单，甚至做市场的同事也可以看到有多少营销活动电子邮件正在发往客户。如果你想要自动化这些任务的话，则可以求助于 REST API，并通过 curl 来调用。由于你得到的应答是 JSON 对象，因而很容易将它集成到当前的工具和语言中。最后，如果你想要导出服务器信息，但又不想手工构造 HTTP 请求之后通过 curl 发送的话，就可以求助于 rabbitmqadmin 脚本来获得更友好的格式化输出，帮助你管理和监控 RabbitMQ。

在下一章中，我们会更深入介绍 REST API，将讲解如何自动化多个管理任务（例如用户和 vhost 配置）。先做个热身吧，因为我们马上就要开始编写 Python 代码了。

9 使用REST API控制Rabbit

本章要点

- Rabbit REST API 的限制和功能
- 用代码来管理Rabbit权限
- 访问消息通信数据统计和计数器
- 自动化创建用户和虚拟主机

到目前为止，你一直受限于应用程序或者脚本来配置 RabbitMQ 服务器。当然你可以编写代码运行 `rabbitmqctl`，然后尝试“倒腾”输出的结果。但是这是个脆弱的解决方案，一旦 Rabbit 团队决定修改 `rabbitmqctl` 的输出，那么它就无法运行了。事实上，`rabbitmqctl` 和 Management Web UI 两者都是为与人交互而设计的。所以，当你想使用像 Chef、Puppet，甚至 CFEngine 这样的工具，来自动化 RabbitMQ 服务器的开发时，该怎么做呢？同时，如何满足监控 RabbitMQ 的基本需求呢？你应该如何编写健康检测脚本来监控 Rabbit，而非内嵌到 RabbitMQ 内部机制中的可编程窗口呢？

如果你是从头到尾顺序阅读本书的话，就已经对 RabbitMQ Management 插件非常熟悉了（参见第 8 章的安装说明）。它提供了一个基于 Web 的访问页面，允许

你通过浏览器来管理和控制 RabbitMQ 服务器。通过内置的 Web UI，管理员可以完成各种任务，包括创建用户和 vhost、查看队列数据统计和总体配置等。当你安装了 Management 插件后，则不仅获得了 Web UI，同时还免费拥有了 REST 化的 Web API。该 API 为应用程序和脚本提供了与 Web UI 或 `rabbitmqctl` 完全相同的功能。

当 Rabbit 总部发布 Management 插件时，这不仅给了开发人员人性化的 Web UI，同时也包含了一个 REST 化的 Web 管理 API。这个 API 使用简单的、语言中立的、Erlang 无关的方式来配置和监控运行中的 Rabbit 节点。“Erlang 无关”这一点尤其重要。虽然 Erlang 为 RabbitMQ 的分布式、可扩展性和稳定性提供支持，但是你可能想要从没有安装 Erlang 的系统上来控制 Rabbit。幸运的是，RabbitMQ 的 Management API 使用 HTTP 协议进行通信，因此任何编程语言或者脚本语言只要有 HTTP 客户端库的，都能进行通信。

> **注意**：以下内容针对的是那些不熟悉 REST 这个术语的读者。REST 表示的是 Representational State Transfer（表述性状态转移）。它描述了基于 HTTP API 的约定，将你正在修改的项目以及该项目上的动作状态编码为 URL。举例来说，你可以用一个非 REST 化的 API，像 `http://my-api.com/calls` 这样，然后将需要修改的项目和动作等详情置于请求正文之中。但是如果你想要对 API 服务器日志做任何类似数据挖掘的话，就会看到日志文件里全部都是 `/calls` 的请求列表。这一点用处都没有。当你按照 REST 约定的方式来组织 API 时，那么 URL 会像这样：`http://my-api.com/item`，然后使用像 POST、PUT 和 DELETE 这样标准的 HTTP 动作来创建、修改或者删除项目。现在日志中填满了有用的信息，显示了用于操作的项目，以及该项目之上的动作。更多关于 REST 的信息，请参阅 http://en.wikipedia.org/wiki/Representational_State_Transfer。

在我们使用 Management API 编写程序之前，需要先介绍 API 允许（以及禁止）你做什么。一旦掌握了这些之后，你就能为 API 客户端创建访问凭证，然后使用该凭证访问 RabbitMQ 内部数据统计，并在 Rabbit 服务器上添加用户和虚拟主机。让我们马上看看 Management API 能做些什么吧！

9.1 能用RabbitMQ REST API做什么

API 是完全 REST 化的，理解这一点很重要。因此操作的项目名称总是会包含在 URL 内。举个例子，如果你想要查看虚拟主机 oak 下名为 branches 的队列数据统计的话，就可以构造如下 URL 并以 HTTP GET 请求的方式发送给服务器：http://localhost:55672/api/queues/oak/branches[1]。如果使用 cURL 来发送请求的话，你会看到如下所示：

```
$ curl -i -u guest:guest
http://localhost:55672/api/queues/oak/branches
HTTP/1.1 200 OK
Server: MochiWeb/1.1 WebMachine/1.7 (participate in the frantic)
Date: Tue, 05 Jul 2011 22:55:25 GMT
Content-Type: application/json
Content-Length: 739
Cache-Control: no-cache

{"memory":9136,"messages":0,"consumer_details":[],
 "idle_since":"2011-7-516:55:1","exclusive_consumer_pid":"",
 "exclusive_consumer_tag":"","messages_ready":0,
 "messages_unacknowledged":0,"messages":0,"consumers":0,
 "backing_queue_status":{"q1":0,"q2":0,
...
```

> **注意**：cURL 让你可以在命令行上手工发送和接收 HTTP 请求。在大多数 Linux/UNIX 发行版本上都能找到它。也可以直接在 http://curl.haxx.se/download.html 进行下载。

喔，除了简单易读的 HTTP 头之外，下面那一大块杂乱的内容是什么呢？如果你看一下 Content-Type 头的话，就会发现应答内容被编码成了 application/json。JSON 代表了 JavaScript Object Notation，可以替代 XML 来编码数据（请参阅 http://en.wikipedia.org/wiki/JSON 来获得更深入的理解）。JSON 允许你使用编程语言能够理解的结构，例如哈希表、数组、字串和整型来编码数据。如果你懂 JavaScript 的话，则可能已经明白了该 API 返回的是哈希表格式的队列数据。举例来说，哈希表中 memory 元素指的是队列当前消耗的内存大小（以字节为单位）。当使用 Management API 时，每一次调用要么返回空的内容（创建或者删除条目的动作）；要么返回一个 JSON 格式的哈希表，包含了你请求的数据（列表或者展现条目的动

---

1 对于本章示例来说，RabbitMQ运行在localhost上；如果没有修改Management插件的话，默认监听端口是55672。

作）。

与 API 请求返回的数据同样重要的是用来发起请求的 HTTP 动作。此处使用的是 GET 请求。RabbitMQ API 将 GET 解释为“给我展示位于 /api/queues/oak/branches 的项目的详情或设置。”除了 GET 之外，你也可以使用 POST、PUT 或者 DELETE。POST 和 PUT 用来创建条目，而 DELETE 用来删除条目。POST 和 PUT 不可互换使用。某些 API 调用需要 POST 来创建条目，而另一些则使用 PUT（查阅本节末尾的参考链接，就能明白什么地方该用 PUT，什么地方该用 POST 了）。假设该 branches 队列尚不存在，你想用 API 来创建它。通过使用和之前一样的 URL，但是将动作更改为 PUT，就能将请求从获取队列详情更改为创建队列：

```
$ curl -i -u guest:guest -X PUT -H
"Content-Type: application/json" \
 -d '{"auto_delete": false, "durable": false}' \
http://localhost:55672/api/queues/oak/branches

HTTP/1.1 204 No Content
Server: MochiWeb/1.1 WebMachine/1.7 (participate in the frantic)
Date: Tue, 05 Jul 2011 23:24:46 GMT
Content-Type: application/json
Content-Length: 0
```

好吧，你并非仅仅将动作更改为 PUT。你同时还将 JSON 哈希表 '{"auto_delete": false, "durable": false}' 添加到了正文中，同时添加了 ContentType 头，这样 API 会知道正文中的数据是 JSON 编码的。请求正文告诉 API，关闭 auto_delete 和 durable 标志。这两个参数和之前本书中 Python 示例创建队列时指定的 auto_delete 和 durable 标志是一样的。事实上，和 Python AMQP 示例一样，API 需要在创建队列时指定标志。如果你将正文留空的话，就会收到一个 API 错误。不管用 API 创建队列还是设置权限，每当使用 PUT 或者 POST 动作的时候，都需要将函数参数编码为 JSON 格式的哈希表，然后添加到请求正文中。

现在你能创建队列并查看它们的数据统计了，但是你还能做什么呢？以下是一些 API 能做的其他事情：

- 查看当前链接列表和详情——/api/connections
- 下载或者上传完整的 RabbitMQ 服务器配置，包括队列、交换器和绑定——/api/all-configuration
- 列出集群中的所有节点（以及它们的数据统计）——/api/nodes

- 创建或者查看 RabbitMQ 用户——/api/users/<user>
- 查看或者创建虚拟主机——/api/vhosts/<vhost>
- 为用户设置权限——/api/permissions/<vhost>/<user>

Management API（以及插件）总是在不断地增强，这里提到的只是众多 API 提供的功能中的一小部分。你可以通过用浏览器访问 http://localhost:55672/api 来查看目前大多数（以及完整的）API 列表和支持的 HTTP 动作。话不多说！让我们创建一些凭证，以便你能使用 API 来编写脚本！

9.2 对客户端授权访问

你可能注意到了使用 cURL 发起的两个 API 请求都将 `-u guest:guest` 作为参数。就像通过 AMQP 访问 RabbitMQ 的应用一样，脚本使用普通的 RabbitMQ 用户名和密码进行认证获取权限（使用 HTTP Basic 认证），从而访问 Management API。举个例子，如果你有一个监控脚本仅能检查队列数据统计的话，就可以创建一个新的 Rabbit 用户，并将该用户在虚拟主机上的权限设置为：

- Read: .*
- Write: (empty)
- Config: (empty)

这样设置将允许该脚本（以及知道该脚本的 Rabbit 用户名和密码的人）只能对队列进行监控，而不能发布消息或者更改配置。

那么如何为 API 访问创建用户呢？这很简单：通过 `rabbitmqctl` 创建用户，并将 `admin` 属性设置为 true 即可。让我们创建一个名为 `monitor` 的用户，脚本可以使用该用户来监控默认（/）虚拟主机（但是不能写或者更改任何东西）的数据统计了。在 RabbitMQ 目录下运行：

```
$ sbin/rabbitmqctl add_user monitor monitorMe
Creating user "monitor" ...
...done.
$ sbin/rabbitmqctl set_admin monitor
Setting administrative status for user "monitor" ...
...done.
$ sbin/rabbitmqctl set_permissions -p / monitor "" "" ".*"
```

```
Setting permissions for user "monitor" in vhost "/" ...
...done.
```

就是这么简单！set_admin命令至关重要。如果没有为用户设置admin标记的话，无论它的权限如何设置，都无法访问API。最后一条命令（set_permissions）赋予monitor用户在默认（/）虚拟主机下完整的读权限，但没有赋予配置或者写权限。在准备好了monitor用户之后，你可以开始编写第一个API脚本来查看队列数据统计了。

9.3 访问数据统计

在日常工作中，总要反复查看特定队列上有多少条消息。某些时候是为了调试新的应用程序；另一些时候则是为了监控产品环境中，等待消费的消息和投递到消费者但仍未确认的消息之间的比率（第二个度量标准对于发现导致消费者崩溃的始作俑者来说非常有用）。虽然你可以使用rabbitmqctl来列出队列的消息总数，但是这有两个主要的缺点：

（1）rabbitmqctl只能在那些安装了Erlang并且和RabbitMQ服务器拥有相同Erlang cookie的计算机上才能运行。

（2）rabbitmqctl只会向你展示队列的消息总数。它不会区分等待确认的已投递消息和等待消费的消息。

听起来对Rabbit Management API来说是个完美的工作！你将使用Python内建的httplib和json库来和Rabbit进行通信，并且对请求和响应进行编码/解码。由于你想要从命令行上运行数据统计查询脚本，因此需要对命令行参数进行解析，如下列清单所示。

清单9.1 queue_stats.py——获取初始设定

```
import sys, json, httplib, urllib, base64

if len(sys.argv) < 6:
    print "USAGE: queue_stats.py server_name:port auth_user " + \
        "auth_pass VHOST QUEUE_NAME"
    sys.exit(1)
```

❶ 有效参数数目

```
server, port = sys.argv[1].split(":")
username = sys.argv[2]
password = sys.argv[3]
vhost = sys.argv[4]
queue_name = sys.argv[5]
```

用参数给内存变量❷赋值

除了程序名称之外，该工具还接收五个参数：服务器名称/端口（以host:port的方式）、用于API认证的用户名、用于API认证的密码、队列所在虚拟主机的名称，以及你想要查看数据统计的队列名称。在你验证了❶必要参数的最小数目后，你将这些参数赋值❷给了内存变量。唯一不寻常的一点是：server, port =sys.argv[1].split(":")。由于传入的Rabbit服务器的主机名和端口是由“:”分隔（localhost:55672）的单一参数，因此你将该参数拆分成为主机名和端口号两部分。split命令将参数在“:”处进行切分，并返回包含各个部分的列表（例如，"localhost:55672"将拆分为["localhost", "55672"]）。分开的部分在之后会被分别赋值给server和port这两个变量。

在解析参数之后，需要构造请求。所有API队列操作都位于/api/queues路径下。所以，如果想要访问特定队列的话，你只需对该路径进行扩展，指明包含该队列的虚拟主机和队列名称即可：/api/queues/<vhost>/<queue_name>，如下列清单所示。

清单9.2 queue_stats.py——构造请求

```
vhost = urllib.quote(vhost, safe='')
queue_name = urllib.quote(queue_name, safe='')
path = "/api/queues/%s/%s" % (vhost, queue_name)
 method = "GET"
```

❶构造API路径

设置请求方法

你可能注意到了，在将虚拟主机名和队列名称放入请求路径前，需要对它们进行转义❶。如果不对虚拟主机进行转义的话，那么指定默认虚拟主机（/）会引发一个API错误。因为服务器会将“/”视为一个路径分隔符。转义之后，请求路径/api/queues///test_queue就会变成/api/queues/%2F/test_queue，API服务器就能理解了。最后，你将HTTP方法设置为GET，因此服务器会知道你想要获取队列的详情，而不是创建一个新队列。在设置了请求路径和方法后，你准备好将请求发送到服务器，如下列清单所示。

清单 9.3 queue_stats.py——发送 API 请求

```
conn = httplib.HTTPConnection(server, port)
credentials = base64.b64encode("%s:%s" % (username, password))
headers = {"Content-Type" : "application/json",
           "Authorization" : "Basic " + credentials}
conn.request(method, path, "", headers)
response = conn.getresponse()
if response.status > 299:
    print "Error executing API call (%d): %s" % (response.status,
                                                 response.read())
    sys.exit(2)
```

❶ 连接到API服务器
❷ 用Base64编码安全凭证
❸ 设置HTTP头
❹ 发送请求
❺ 接收应答

这段脚本首先建立了到 API 服务器的 HTTP 连接❶。此时，服务器等待 API 请求和授权凭证。由于 Rabbit API 使用 HTTP Basic 认证，因此你需要将用户名和密码封装为❷ `username:password` 格式并进行 Base64 编码[2]。然后，你需要创建一个字典（哈希表）用来包含 HTTP 头❸。其中之一是 `Authorization` 头，它包含了基于 Base64 编码的凭证，添加在 ASCII 字符串 `"Basic"` 之后。另一个同样重要的是 `Content-Type` 头，因为它让 API 服务器知道请求正文（如果有的话）是以 JSON 格式编码的。由于这里只发送 `GET` 请求，因此请求正文总是空的。但设置 `Content-Type` 是一个好习惯，以免在需要的时候（`PUT` 或者 `POST` 请求）忘记。最后，你将请求发送❹到 API 服务器并接收响应❺。如果在完成请求之前发生问题的话，API 服务器会将 HTTP 响应代码设置为 4xx 或 5xx。任何高于 299 的响应代码都非正常情况，所以检测代码时应小于 299 ❺。如果发生错误的话，需要将错误报告给用户并退出程序；否则就解析响应并将队列详情展现给用户，如下列清单所示。

清单 9.4 queue_stats.py——解析并展现队列数据统计

```
payload = json.loads(response.read())

print "\tMemory Used (bytes): " + str(payload["memory"])
print "\tConsumer Count: " + str(payload["consumers"])
print "\tMessages:"
print "\t\tUnack'd: " + str(payload["messages_unacknowledged"])
print "\t\tReady: " + str(payload["messages_ready"])
print "\t\tTotal: " + str(payload["messages"])
sys.exit(0)
```

❶ 对响应进行解码
❷ 展现队列数据统计

---

2 Base64是一种将文本和二进制数据编码为ASCII字符的方法。HTTP Basic认证规范要求 `username:password` 在发送之前需要进行Base64编码。有关Base64的更多信息请参阅http://en.wikipedia.org/wiki/Base64。

由于 API 服务器总是以 JSON 哈希表的格式返回信息，因此第一步要做的是将响应内容用 JSON 进行解码❶。这一步将 JSON 编码的哈希表转换成了编程语言支持的哈希表类型。对于 Python 来说，这意味着响应被转换成了一个 Python 字段。响应被编码为 JSON 格式的美妙之处在于一旦解码完成，你就可以像其他字典（哈希表）那样访问 resp_payload 中的字段了。举例来说，获取队列占用的内存数量就只需要访问❷ resp_payload 字典的 memory 元素即可[3]。

那么，程序运行起来是什么样子呢?

```
$ python queue_stats.py localhost:55672 guest
guest / test
'test' Queue Stats
-----------------
    Memory Used (bytes): 9104
    Consumer Count: 3
    Messages:
        Unack'd: 3
        Ready: 4
        Total: 7
```

从输出可以看到，默认（/）虚拟主机内的 test 队列消耗了 9104 字节的内存，拥有三个消费者，并且包含了总共七条消息。除了这七条消息之外，你还能看到其中的三条消息已经发送给了消费者，并等待确认，同时另外的四条正等待发送给下一个消费者。只用了不到 60 行的代码，并且你可以从基础设施中的任何地方进行访问！这就是 Rabbit Management API 的强大之处。它使得你可以在任何能够访问网络的地方监控并控制 Rabbit。到目前为止我们所做的是读取数据统计，这对于控制 RabbitMQ 还不够。让我们对 RabbitMQ 服务器的配置做些改动吧。

9.4 自动化vhost和用户配置

在使用 Management API 之前，部署 RabbitMQ 中最麻烦的就是，自动化创建虚拟主机和用户以满足应用的需要了。像 Chef 或 Puppet 这样的自动化部署工具运行的服务器，通常和应用程序需要通信的 RabbitMQ 服务器不相同。当只能用 rabbitmqctl 来创建用户和虚拟主机的时候，问题就来了。因为这意味着你必须在所有应用服务器上安装 Erlang 和 rabbitmqctl。如果可以在运行应用的那

3 resp_payload字典中还有许多其他数据统计，这里展示的只是其中一部分而已。其中还包含了持久化消息数量、等待确认消息数量、队列中消息的平均进出速率等。在清单中的sys.exit(0)之前添加print repr(resp_payload)用以查看所有的字段。

台服务器上就能创建用户和 vhost，而不需要将 Erlang 安装到每一个服务器上（并且需要将所有的 Erlang cookie 进行同步）的话，岂不妙哉？幸好我们活在 Rabbit Management API 的时代！你可以使用 Management API 编写一个命令行脚本，它能够创建、删除、展示以及列出用户列表（另外做些简单的扩展就能配置 vhost 了）。而且不需要安装 Erlang 或者 Erlang cookie！

由于我们已经介绍了如何连接、认证和发送基本 API 请求，下面将要专注于为何创建和删除用户（或者 vhost）的请求互不相同。让我们从 `user_manager.py` 脚本的完整内容开始，如下列清单所示。

清单 9.5 user_manager.py——RabbitMQ 用户管理

```
import sys, json, httplib, base64

if len(sys.argv) < 5:
    print "USAGE: user_manager.py server_name:port " + \
          "auth_user auth_pass",
    print "ACTION [PARAMS...]"
    sys.exit(1)

server, port = sys.argv[1].split(":")
username = sys.argv[2]
password = sys.argv[3]
action = sys.argv[4]

if len(sys.argv) > 5:
    res_params = sys.argv[5:]
else:
    res_params = []

base_path = "/api/users"

if action == "list":
    path = base_path
    method = "GET"
if action == "create":
    path = base_path + "/" + res_params[0]
    method = "PUT"
if action == "delete":
    path = base_path + "/" + res_params[0]
    method = "DELETE"
if action == "show":
    path = base_path
    method = "GET"

json_args = ""
if action == "create":
    json_args = {"password" : res_params[1],
                 "administrator" : json.loads(res_params[2])}
```

❶ 分配参数

❷ 构造API路径

❸ 构造JSON格式参数

```
        json_args = json.dumps(json_args)

    conn = httplib.HTTPConnection(server, port)
    credentials = base64.b64encode("%s:%s" % (username, password))
    conn.request(method, path, json_args,
                 {"Content-Type" : "application/json",
                  "Authorization" : "Basic " + credentials})
    response = conn.getresponse()
    if response.status > 299:
        print "Error executing API call (%d): %s" % (response.status,
                                                     response.read())
        sys.exit(2)
    resp_payload = response.read()
    if action in ["list", "show"]:
        resp_payload = json.loads(resp_payload)

        if action == "list":
            print "Count: %d" % len(resp_payload)
            for user in resp_payload:
                print "User: %(name)s" % user
                print "\tPassword: %(password_hash)s" % user
                print "\tAdministrator: %(administrator)s\n" % user
        if action == "show":
            print "User: %(name)s" % resp_payload
            print "\tPassword: %(password_hash)s" % resp_payload
            print "\tAdministrator: %(administrator)s\n" % resp_payload
    else:
        print "Completed request!"

    sys.exit(0)
```

发送API请求 4

解析并展示响应内容 5

处理list结果 6

处理show结果 7

create和delete动作的请求返回结果为空 8

和队列数据统计脚本相比，你验证了❶命令行参数并将其赋值给各内存变量。到目前为止唯一不同的地方在于参数变多了。之前用于表示 vhost 和队列名称的那个参数，现在用来表示采取的动作（create，delete，list，show）以及动作相关的参数了。举个例子：

```
$ python user_manager.py localhost:55672 guest guest \
create myuser password true
```

这会连接到 localhost API 服务器，使用 guest 作为用户名和密码，并创建名为 myuser 的用户，密码为 password，并且是一位管理员（true）。如果你将 true 更改为 false 的话，那么该用户将不会成为管理员。

下一步开始构造请求路径❷。所有 API 请求的基本路径是 /api/users。你将用户名（在 res_params 数组中的下标 0 处）添加到基本路径之后：

```
base_path = "/api/users"
```

```
if action == "list":
    path = base_path
    method = "GET"
if action == "create":
    path = base_path + "/" + res_params[0]
    method = "PUT"
if action == "delete":
    path = base_path + "/" + res_params[0]
    method = "DELETE"
if action == "show":
    path = base_path + "/" + res_params[0]
    method = "GET"
```

在这里，你可以清楚地看到 API 是如何解释相同请求路径上不同的 HTTP 动作的。如果将方法设置为 PUT，那么 API 服务器会根据 /api/users/<username> 来创建用户。沿用同样的请求路径，但是将方法设置为 DELETE 的话，会导致用户被删除。同样地，API 服务器会将 GET 解读为将用户详细信息以 JSON 哈希表的格式返回。唯一古怪的是列出所有用户。你在 /api/users 上发送 GET 请求而不是在 /api/users/<username> 上（这里会明确指明用户名）发送请求。由于请求路径不指明任何特定的用户，因此 API 服务器会返回 RabbitMQ 服务器上所有的用户（服务器返回哈希表列表，每个哈希表代表了一位用户的详细信息）。

在构造完路径之后，你需要决定该请求是否应该创建用户❸。如果是的话，则需要构建一个 JSON 编码的哈希表来包含新用户的密码和管理员状态。

```
json_args = ""
if action == "create":
    json_args = {"password" : res_params[1],
                 "administrator" : json.loads(res_params[2])}
    json_args = json.dumps(json_args)
```

用于创建用户的哈希表需要两个字段：password 和 administrator。password 包含了一个纯文本密码；而 administrator 是一个布尔类型，需要设置为 true 或者 false。请求构造完毕后，你就可以像在队列数据统计脚本中做的那样进行连接并发送❹请求了。唯一不同之处在于，当动作为创建用户的时候，需要为请求指定正文。

一旦请求发送出去之后，并且没有发生任何错误的话，你读取❺响应并继续运行程序。如果该动作是创建或者删除用户的话❽，那就没有返回内容了，所以只需

打印一个确认消息，说明请求已经成功了。如果请求动作是 list 或者 show 的话，那么你就要用 JSON 对响应进行解码。在解码之后，你就有了本地 Python 数据结构。如果用户请求的是用户列表的话，那么将返回哈希表数组。因此你需要对用户列表进行迭代❻，并打印每位用户的 name、password_hash 和 administrator 状态[4]。

```
if action == "list":
        print "Count: %d" % len(resp_payload)
        for user in resp_payload:
            print "User: %(name)s" % user
            print "\tPassword: %(password_hash)s" % user
            print "\tAdministrator: %(administrator)s\n" % user
```

如果是展示特定用户的话，那么响应结果不会是哈希表数组，而是单一的哈希表。除了❼包含了用户的 name、password_hash 和 administrator 外，和你在队列数据统计脚本中看到的相似。

```
if action == "show":
        print "User: %(name)s" % resp_payload
        print "\tPassword: %(password_hash)s" % resp_payload
        print "\tAdministrator: %(administrator)s\n" % resp_payload
```

现在你拥有了功能丰富的实用程序，可以在任何安装了 Management 插件的 RabbitMQ 服务器上创建、删除、展示以及列出用户列表。让我们看看，如何使用该程序先创建用户，再展示用户：

```
$ python user_manager.py localhost:55672 guest guest \
  create newuser newpass true
Completed request!

$ python user_manager.py localhost:55672 guest guest \
  show newuser
User: newuser
    Password: o/ZEH9Z86FNUtzu2MzNlmDSTDFE=
    Administrator: True
```

在产品环境中，你可能想要为脚本添加严格的参数验证。但事实上你可以原样使用它来管理用户的创建。事实上，我们在部署应用时，就使用了相似的脚本来自

4 你可能注意到我们使用的是Python的字符串格式化功能。举例来说，"User: %(name)s" % user告诉Python创建一个新的字符串，并用值user["name"]替换字符串中%(name)s的位置。你可以使用语法"User: " + user["name"]来获得相同的结果。

动化创建 RabbitMQ 用户。通过构造这段脚本，你不仅学习了如何使用 Management API 来展示条目，还学习了如何展示条目列表，以及如何创建和删除这些条目。你可以将这些概念应用到处理其他条目 / 资源类型（用户、队列、交换器、连接、权限等）。本章附带了一份脚本，名为 `user_vhost_manager.py`。它扩展了用户管理脚本，添加了对创建 vhost 的管理，以及对用户或者 vhost 权限的操作。

9.5 总结

在我们开始本章之前，手工管理 Rabbit 服务器的方式无非是 `rabbitmqctl` 和 Management Web UI。你无法编写自动化脚本或者实用程序来和 Rabbit 集成，用于管理服务器配置或者监控内部状态。但是通过学习 Rabbit Management API 的工作机制，你现在可以构建工具，用来监控队列状态并根据需求管理用户了。你不再受手工管理 RabbitMQ 服务器的摆布，因为现在你能编写实用程序，通过使用 Management API 来自动化 Rabbit 配置，定制适合组织的过程。你可以尽情使用 Management API，构建 Rabbit 管理工具的唯一限制就是你的想象力。Rabbit Management API 还可以支持自动化监控 RabbitMQ 的健康状态。在理解了如何使用 API 构建工具之后，你能更深入地利用这些技能来加强对 Rabbit 基础设施的监控，并确保它们处于最佳状态。

10 监控

本章要点

- 编写Nagios健康检测的基础
- 使用AMQP和REST API来监控Rabbit内部状态
- 确认Rabbit可用并且能够进行响应
- 观察队列状态以尽早检测消费者问题
- 检测消息通信结构中不合需求的配置更改

你的 RabbitMQ 服务器运行正常，同时你那风靡全球的遛狗应用正带来数以千计的订单。这一切看起来棒极了，直到你突然接到了电话：客户在使用 Web 应用时遇到了错误，并且订单处理流程完全停止了。RabbitMQ 服务器宕机了，而且更糟糕的是看起来服务器已经宕机了好几个小时了。你真心希望能在 RabbitMQ 一宕机就能主动收到通知，而不是事发之后由客户来告诉你，因为那时候你已经损失好几千美元了。将 Rabbit 设置成高可用性，然后使用它来编写应用，这还不够。之前的悲剧仍然会重复发生。如果你希望运维能够足够可靠的话，那么就需要监控 Rabbit，以确保它正常运行，并且你发送给它的消息都能如期消费。采取适当的监控措施，你不仅能在事情变糟时及时发现（譬如 Rabbit 服务器命悬一线时），而且还能及时发现消息通信基础设施是否正饱受煎熬——这就能避免让你的客户失望，同时守住

了你的底线。举个例子，如果遛狗订单队列突然之间有 1000 万条消息在等待消费，而平时正常情况下只有 1000 条消息，这时候你能够及时收到通知的话，那不是很棒吗？或者订单队列的 durable 设定突然之间变成了 transient，在遇到断电并导致所有订单都消失之前，你能及时知道的话，那也很不错吧？在本章中，你将学习如何编写程序来监控 Rabbit 以及消息通信基础设施的方方面面。然后你就能将这些监控程序植入监控和告警框架之中，因而当消息通信偏离轨道时你能及时收到告警通知。让我们从如何使用 AMQP 和 REST API 来监控 RabbitMQ 的内部状态开始吧。

10.1 监控RabbitMQ：密切关注你的warren

监控 RabbitMQ 并不只是确保端口 5672 是开启的并能接收 TCP 连接而已。对于像 Rabbit 这样复杂的系统，如果你能够模拟 AMQP 客户端来确保连接之后获取信道的话，才算那么回事儿。如果你能使用 REST API 来找出是否所有构成 Rabbit 的 Erlang 部件都正常运行，并且它们之间能正确通信的话，这就非常不错了。为了实现上述功能，你必须理解如何编写一个监控系统可以理解的健康检测程序。在这里，系统指的是 Nagios。让我们看看 Nagios 是什么，以及一个健康检测该实现哪些功能。

10.1.1 为 Nagios 编写健康检测

当今有许多不同的商业和开源监控框架可供选择。最流行的其中之一是 Nagios。它可以从 http://nagios.org 上免费获取。同时它拥有一个灵活的 API，用任何语言来编写自己的健康检测程序都十分简单。此外，许多其他开源监控框架（像 Zenoss 和 Zabbix）也支持 Nagios 健康检测 API，这使得针对 Rabbit 的健康检测程序可以被除了 Nagios 的其他监控系统使用。那么什么是 Nagios 健康检测程序，它又是如何运作的呢？

Nagios 健康检测是一个独立的程序，它在运行时监控服务并在程序终止运行时退出代码来指示服务的健康状况。从技术上来讲，你甚至不需要 Nagios 来运行健康检测——你可以在任何时候通过命令行执行并手工观测输出。Nagios 健康检测可以用任何语言编写，可以是 Python 程序，甚至是 BASH 脚本。检测程序需要将可读状态打印到 STDOUT 上，并且返回下列四种整型退出代码之一：

- *0*—OK——接收检测的服务工作正常，并且各项指标都处于通过命令行参数设定的阈值之内。
- *1*—WARNING——服务运行处于退化状态（或者说是遇到了问题），但是这个问题并不紧急。举个例子，假设你正在监控内存使用情况。如果你将健康检测配置了 2GB 的 warning 阈值，以及 4GB 的 critical 阈值的话，任何内存使用处于 2 ~ 4GB 之间的情况都会返回一个 WARNING 退出代码。
- *2*—CRITICAL——服务关闭了、无响应，并且 / 或者超过了受监控的临界度量阈值。还是拿内存使用情况这个例子来说明，当内存使用超过 4GB 时，健康检测会返回一个 CRITICAL 退出代码。
- *3*—UNKNOWN——从技术上来讲，这意味着服务的状态或者监控的度量值无法确定。举例来说，如果健康检测程序正监控队列上的消息总数，但是它无法连接上服务器，则它会返回 unknown 状态码而非 critical 状态码。仅当你无法取样度量值的当前状态时，返回 unknown 状态码才是有意义的。如果需要监控的度量值就是目标是否可连接，而你又无法连接上的话，那么当然应该返回 critical 而不是 unknown 了。

现在你理解了 Nagios 期望从健康检测程序中获得什么之后，让我们这就来编写一个吧，如下列清单所示。虽然你可以使用任何语言来编写健康检测程序，但是你将再次使用自己在第 4 章安装的 Python 和 Pika AMQP 库来编写示例程序。第一个健康检测程序会非常简单，因为它不会检测任何东西。它会接收 warning、critical、unknown 或者 ok 作为参数，并且以 Nagios 状态码退出。

清单 10.1 nagios_check.py：返回 Nagios 状态代码的健康检测程序

```
import sys, json, httplib, base64

status = sys.argv[1]

if status.lower() == "warning":
    print "Status is WARN"
    exit(1)
elif status.lower() == "critical":
    print "Status is CRITICAL"
    exit(2)
elif status.lower() == "unknown":
    print "Status is UNKNOWN"
    exit(3)
else:
    print "Status is OK"
    exit(0)
```

返回请求的Nagios状态代码

如果你运行 python nagios_check.py critical，则应该会在 STDOUT 上看到一条可读消息，并且退出状态码为 2：

```
> python nagios_check.py critical
Status is CRITICAL
> echo $?
2
```

棒极了！你的健康检测程序返回了正确的状态消息，而且 echo $? 告诉你退出码是正确的 critical 状态（2）。如果 Nagios 运行这段健康检测程序的话，你会得到一个告警，并显示为 Status is CRITICAL。虽然 Nagios 不理解这段消息的真正含义，但它理解退出码 2 表明处于 critical 状态。接下来我们所有的健康检测都会基于该示例之上，并逐步添加逻辑代码来检测在线服务，并且也能接收命令行参数，从而健康检测就能知道该去监控哪个服务以及 critical 或者 warning 阈值的组成了。

在继续构建健康检测来查看 RabbitMQ 是否运作以及是否能够构造信道之前，我们需要先来了解 Nagios。虽然你构造的健康检测可以被 Nagios 使用，但是我们不会讨论如何配置 Nagios 来使用它们。这样做可以让我们专注于监控 Rabbit 的方方面面。如果你想要学习更多关于 Nagios 的知识以及如何安装的话，请参阅文档站点 http://www.nagios.org/documentation。

话不多说。让我们这就来检测你的 RabbitMQ 服务器是否运作正常、足够健康！

10.1.2 使用 AMQP 模拟检测来确认 RabbitMQ 是否运行

不用编写一行代码，你就能使用大多数监控系统附带的 TCP 健康检测程序，通过 TCP 连接测试 Rabbit 是否能在端口 5762 上响应。虽然这会告诉你 RabbitMQ 守护进程是否在运行，但你却不知道它是否正常运作。例如，RabbitMQ 内存不足怎么办呢？守护进程仍然可能完成 TCP 握手，但却没有足够的内存来响应 AMQP 命令。为了能够真正地判断 Rabbit 是否有能力来服务请求，你需要真实地发送 AMQP 命令。因此，让我们来构造一个 AMQP ping 健康检测。当下列任何条件之一为真时，该检测程序会返回一个 critical 状态。

- RabbitMQ 没有响应 TCP 连接。
- 当发送 AMQP 命令时，Pika 在接收到响应之前超时了。
- 当构造 AMQP 信道时，遇到了协议错误。

仅当这些状态检测都为 false 时，健康检测程序才会返回 OK 状态。健康检测代码看起来很像首个 Hello World 消费者程序的简化版本。

清单 10.2 amqp_ping_check.py

```
import sys, pika

EXIT_OK = 0                                          ← ❶ Nagios状态码
EXIT_WARNING = 1
EXIT_CRITICAL = 2
EXIT_UNKNOWN = 3

server, port = sys.argv[1].split(":")                ← ❷ 解析命令行参数
vhost = sys.argv[2]

username = sys.argv[3]
password = sys.argv[4]

creds_broker = pika.PlainCredentials(username, password)
conn_params = pika.ConnectionParameters(server,      ← ❸ 建立到代理服务器的连接
                                        virtual_host = vhost,
                                        credentials = creds_broker)
try:
    conn_broker = pika.BlockingConnection(conn_params)
    channel = conn_broker.channel()
except Exception:

    print "CRITICAL: Could not connect to %s:%s!" % \   ← ❹ 连接失败，返回CRITICAL状态
    (server, port)
    exit(EXIT_CRITICAL)

print "OK: Connect to %s:%s successful." % (server, port)   ← ❺ 连接成功，返回OK状态
exit(EXIT_OK)
```

这些常量代表了 Nagios 期望健康检测程序返回的状态代码❶，之后你开始解析传入的命令行参数❷。由于该健康检测只是简单地验证 RabbitMQ 服务器是否能够响应 AMQP 命令，因此它仅需四个参数即可：RabbitMQ 的服务器名称和端口、用于构造信道的虚拟主机(vhost),以及用于授权访问 vhost 的用户名和密码。等等——总共 5 个参数，不是 4 个啊。为了简化传入的参数数目，我们将主机名和端口号合并成单一参数，以冒号分隔。举例来说，如果 Rabbit 服务器运行在 localhost 的 5672 端口的话，你传入 `localhost:5672` 这样的参数即可。

在传入命令行参数之后，你使用凭证来尝试连接服务器❸，然后构造了一条信道：`channel = conn_broker.channel()`。和先前示例的不同之处在于，当你尝试连接时，不论什么原因抛出异常的话，代码会立即退出，退出代码为 critical ❹(并且打印出无法连接的信息)。在之前的示例中，当发生错误时，你会捕获该错误，

并立即尝试重新连接以保持消费者正常运行。而在该示例中，你的目的就是想要告诉 Nagios 连接失败了，而不是将错误隐藏起来。而且，通过捕获通用 Exception 错误类，任何 Rabbit 服务器相关的问题（包括超时错误）所导致的所有异常都会被错误处理代码捕获，并将 critical 状态（退出代码 2）返回给 Nagios。最后，如果建立连接和信道的时候没有引发任何错误的话，将向 Nagios 返回❺退出代码 0，以表示健康检测成功了。

值得注意的是，在整个过程中，Nagios 都对 RabbitMQ 一无所知。事实上，Nagios 甚至都不知道 RabbitMQ 是什么。Nagios 所知道的一切就是，要么健康检测识别出了严重问题，退出代码为 2；要么所有一切正常运行，退出代码为 0。这就是为 Nagios 编写健康检测程序如此简单的原因，以及为何有如此之多的其他监控框架支持 Nagios 健康检测模型。它们被编写为独立程序，并返回 0~3 之间的退出代码。在构建完健康检测程序之后，让我们用它来测试健康的 RabbitMQ 服务器，看看会发生什么：

```
$ python amqp_ping_check.py localhost:5672 / guest guest
OK: Connect to localhost:5672 successful.
$ echo $?
0
```

太棒了，运行在你本地开发机器上的 RabbitMQ 服务器一切正常，并且能够处理 AMQP 命令。同时根据你的要求，当检测成功的时候，退出代码为 0（echo $? 总是会向你展示最近运行的命令的退出代码）。但是，你该如何确保健康检测程序可以检测出有问题的 RabbitMQ 服务器？为了模拟一个内部 Rabbit 故障，你将使用在处理集群时学到的 rabbitmqctl stop_app 命令：

```
$ rabbitmqctl stop_app
Stopping node rabbit@Phantome ...
...done.
$ python amqp_ping_check.py localhost:5672 / guest guest
CRITICAL: Could not connect to localhost:5672!
$ echo $?
2
```

如预期的那样，基于 AMQP 的健康检测程序成功检测出了故障的 Rabbit，并且返回给监控框架错误代码 2，表示有严重的故障。在有了这个新的健康检测程序之后，你可以轻松地配置监控框架来开启对 Rabbit 服务器的监控，并在服务器无法服务连接请求时及时通知到你。但是如果你期望自己的健康检测程序不仅仅只进行

构造信道这样的测试，而是想要对消息的发布与消费进行完整的测试呢？虽然你可以通过扩展 AMQP 健康检测程序来添加额外的功能，但是如果你已经安装了 Rabbit Management API 的话，你手边就会有更好的选择。让我们看看如何使用 REST API 构造一个健康检测程序来进行完整的生产 / 消费测试，并让健康检测程序知道结果。

10.1.3 使用 REST API 来检测

通过测试 RabbitMQ 是否能够接收新的请求和构造 AMQP 信道，可以用来验证 Rabbit 服务器是否健康。但是我们已经吸取了教训：如果不将过程中的每个部分检测到位的话，万一未检测到的部分发生故障，那就倒霉了。谨记这些，让我们在监控这条路上往前迈一步。我们将检测消息通信的整个过程，向 RabbitMQ 发布消息然后消费该消息，来验证消息被正确地路由了。

虽然你可以通过简单地扩展 AMQP 健康检测程序来对路由过程进行完整的测试，但是检测程序会因此而增添额外的复杂性。因为它需要创建队列，并确保如果健康检测没有完成的话消息不会建立。所幸你有其他选择。随 RabbitMQ Management 插件一同发布的 REST API 的特性之一，就是一个可以内部检测 Rabbit 服务器健康状态的 API。`aliveness-test`，顾名思义，使用三个步骤来验证 Rabbit 服务器是否健康：

（1）创建一个队列来接收测试消息。

（2）用队列名称作为消息路由键，将消息发往默认交换器。

（3）当消息到达队列的时候就消费该消息；否则就报错。

由于检测程序运行在 Erlang 虚拟机内部（随 RabbitMQ 一起），因此它不会受到网络问题的影响。如果在虚拟机外部的话，网络问题可能会阻止你连接到 Rabbit 的端口（`5672`）。虽然使用该 API 来对 Rabbit 进行健康检测意味着可以让你专注于内部消息路由问题，但是这同时意味着这种检测方式无法告诉你防火墙规则是否阻挡了外部消费者（譬如你的遛狗应用）来连接 Rabbit。因此，在实际使用当中，通过先前章节中构建的 AMQP 健康检测与基于 API 的健康检测两者相结合的方式，可以确保对 Rabbit 服务器的全方位监控。特别需要注意的是 `aliveness-test` API 检测过程的实现非常智能，它不会删除创建的队列。这意味着如果你的健康检测程序以非常短的周期重复运行的话，可以避免数以千计的队列元数据事务填满 Mnesia 数

据库。那么如何使用 aliveness-test API 来编写健康检测程序呢？使用 HTTP 以 /api/aliveness-test/<vhost> 的格式来发送 API 请求。其中 <vhost> 指的是虚拟机名称，同时也是 API 用来创建队列的地方。下列清单展示了这段代码。

清单 10.3 api_ping_check.py：针对 RabbitMQ 的基于 REST API 的健康检测程序

```
import sys, json, httplib, urllib, base64, socket

EXIT_OK = 0
EXIT_WARNING = 1
EXIT_CRITICAL = 2
EXIT_UNKNOWN = 3

server, port = sys.argv[1].split(":")
vhost = sys.argv[2]
username = sys.argv[3]
password = sys.argv[4]

conn = httplib.HTTPConnection(server, port)

path = "/api/aliveness-test/%s" % urllib.quote(vhost, safe="")
method = "GET"

credentials = base64.b64encode("%s:%s" % (username, password))

try:
    conn.request(method, path, "",
                 {"Content-Type" : "application/json",
                  "Authorization" : "Basic " + credentials})

except socket.error:
    print "CRITICAL: Could not connect to %s:%s" % (server, port)
    exit(EXIT_CRITICAL)

response = conn.getresponse()

if response.status > 299:
    print "CRITICAL: Broker not alive: %s" % response.read()
    exit(EXIT_CRITICAL)

print "OK: Broker alive: %s" % response.read()
exit(EXIT_OK)
```

❶ Nagios状态代码
❷ 解析参数
❸ 连接到服务器
❹ 构造API路径
❺ 发送API请求
❻ 无法建立连接，返回critical状态
❼ RabbitMQ无响应/无心跳，返回critical状态
❽ RabbitMQ正常运行，返回OK状态

就像在 AMQP 健康检测中做的那样，首要任务是为退出代码设置常量❶，并解析命令行参数❷。该健康检测程序接收和 AMQP 健康检测程序一样的参数，并对 API 服务器的 server 和 port 做了相应的更改（替换了 RabbitMQ 服务器的配置）。和 AMQP 健康检测程序完全不同的地方是，你构建一条到 API 服务器的 HTTP 连接❸，而不是 AMQP 连接。在构建好了到 API 服务器的 HTTP 连接后，你为 aliveness-test 创建了请求路径❹。你将 vhost 附加在路径的结尾处，用于创建测试队列。由于可能会有名为“/”（HTTP 路径分隔符）的 vhost，因此你需要在添

加到路径之前，使用 urllib.quote 对 vhost 进行转义。这里的 safe="" 参数告诉 urllib.quote 方法需要无一例外地转义所有特殊字符（默认情况下，urllib.quote 不会转义“/”字符）。同时，因为你通过 API 来获取信息，而非修改或者创建，所以此处将请求方法设定为 GET。

在创建了 HTTP 连接和请求路径之后，你准备好了编码凭证❺并将请求发送到 API 服务器[1]。如果无法连接上 API 服务器❻的话，程序将返回 EXIT_CRITICAL 作为状态退出代码。由于从技术上来讲，你只知道是 API 服务器宕机了而不是 RabbitMQ 服务器，所以可以返回 warning 或者 unknown 状态。但是由于 API 是 RabbitMQ 的一个插件，所以不太可能 API 服务器宕机了而 RabbitMQ 服务器没有。

假设你能连接上 API 服务器并发送请求，你会接收到一个响应对象。响应对象将 HTTP 状态码存储在 response.status 中，并且可以像访问文件描述符那样通过 response.read() 来访问响应内容。你最关心的就是 HTTP 状态码了。如果 aliveness-test 方法调用成功的话，它会返回 200 的 HTTP 状态码。任何高于 299 的代码要么代表错误，要么就是发送给客户端的额外指令。因此，如果你收到了大于 299 的 HTTP 状态码❼的话，你可以返回 critical 状态（EXIT_CRITICAL）给 Nagios，用于表明健康检测失败了。否则，方法调用成功，并且服务器工作正常❽。由于成功调用返回的正文内容是简单的 JSON 格式字串 {'status' :'ok'}（你熟悉的 HTTP 状态码之一），因此你可以忽略它，也可以将其附加到响应中去。

那么，如果针对本地开发机器运行 API ping 健康检测的话会如何呢？

```
> python api_ping_check.py localhost:55672 / guest guest
OK: Broker alive: {"status":"ok"}
> echo $?
0
```

健康检测程序正确检测出 RabbitMQ 服务器正常运行，并且能够生产和消费消息。但是，如果我们模拟节点故障来响应健康检查的话会发生什么呢？

```
> rabbitmqctl -n rabbit@Phantome stop_app
Stopping node rabbit@Phantome ...
...done.
> python api_ping_check.py localhost:55672 / guest guest
CRITICAL: Broker not alive: <html><head><title>500 Internal
Server Error</title></head><body><h1>Internal Server Error</h1>The
```

---

1 在第9章REST API的介绍中有针对如何构造HTTP Basic认证头的详细介绍。

```
server encountered an error while processing this request:<br><pre>
{exit,{aborted,{no_exists,[rabbit_user,<<"guest">>]}},
      [{mnesia,abort,1},
       {rabbit_misc,dirty_read,1},
       {rabbit_auth_backend_internal,internal_check_user_login,2},
       {rabbit_access_control,'-check_user_login/2-fun-0-',4},
       {lists,foldl,3},
       {rabbit_mgmt_util,is_authorized,3},
       {webmachine_resource,resource_call,3},
       {webmachine_resource,do,3}]}</pre><P><HR>
       <ADDRESS>mochiweb+webmachine
       web server</ADDRESS></body></html>
> echo $?
2
```

喔，一大堆输出。虽然 Nagios 只会读取退出代码（2）来判定健康检测是否失败，但是它会将生成的告警内容包括从 `CRITICAL:...` 到 `...</html>` 全部发送给你。`aliveness-test` 调用返回的内容，以及健康检测程序逐字输出的内容，都是因为 API 无法和 RabbitMQ 进行通信而生成的内部 Erlang 崩溃报告。因此，如果有人关闭了 RabbitMQ 节点，则不仅监控系统能够及时通知你，同时它也会给你一份详细的 Erlang 崩溃报告，你可以用它来追踪为何 Rabbit 会发生这样的问题。

现在你不仅有能力监控 RabbitMQ 是否能接收连接，同时也能检测它是否能成功路由消息。但是如果有人将队列的持久化属性修改为非持久化，从而更容易遭受消息丢失的话，你该怎么办呢？你如何保护 Rabbit 配置免遭危险的修改？要知道，这样的修改通过普通的 RabbitMQ 服务器健康检测是无法察觉到的。不过这也很简单：只要编写一个能够监控队列（或者交换器）配置的健康检测即可。

10.1.4 监控配置文件修改

验证 RabbitMQ 健康运行只是确保消息通信架构可靠性的一部分。同时，你也需要确保消息通信结构配置没有遭受意外修改，从而避免应用消息丢失。举例来说，想象一下努力工作的开发人员 Rolf 正在部署遛狗应用的最新版本。由于你明智地编写了应用来配置所需的队列、交换器和绑定，所以你不必担心由于 RabbitMQ 服务器缺失了消息通信结构中的一部分，会在启动时（应用会创建丢失的部分）崩溃。但是今天早上这份通信结构自动化配置引发了问题，这是因为昨晚 Rolf 解决了 Dog Walker 10.0 最后一个 bug 而工作到很晚，并且不小心犯了个错误。不凑巧的是，Rolf 将应用启动时创建 `walking_orders` 队列的代码删除了。在发现了这个问题

之后，他在将代码提交到产品代码库之前，重新添加了队列声明。问题在于当Rolf重新编写队列声明时，他忘记将队列设置为可持久化了。这可是个大问题，因为在生产环境中如果RabbitMQ服务器遇到了供电故障的话，所有队列上的遛狗订单就都消失不见了。但是由于该队列确实存在并且名称也是对的，因而你不会发现任何问题，直到供电故障时则为时已晚。由于我们总是会成为Rolf这样的人，因此你需要创建健康检测来监控队列配置，在队列发生变化时你能主动收到通知。来看看图10.1吧。

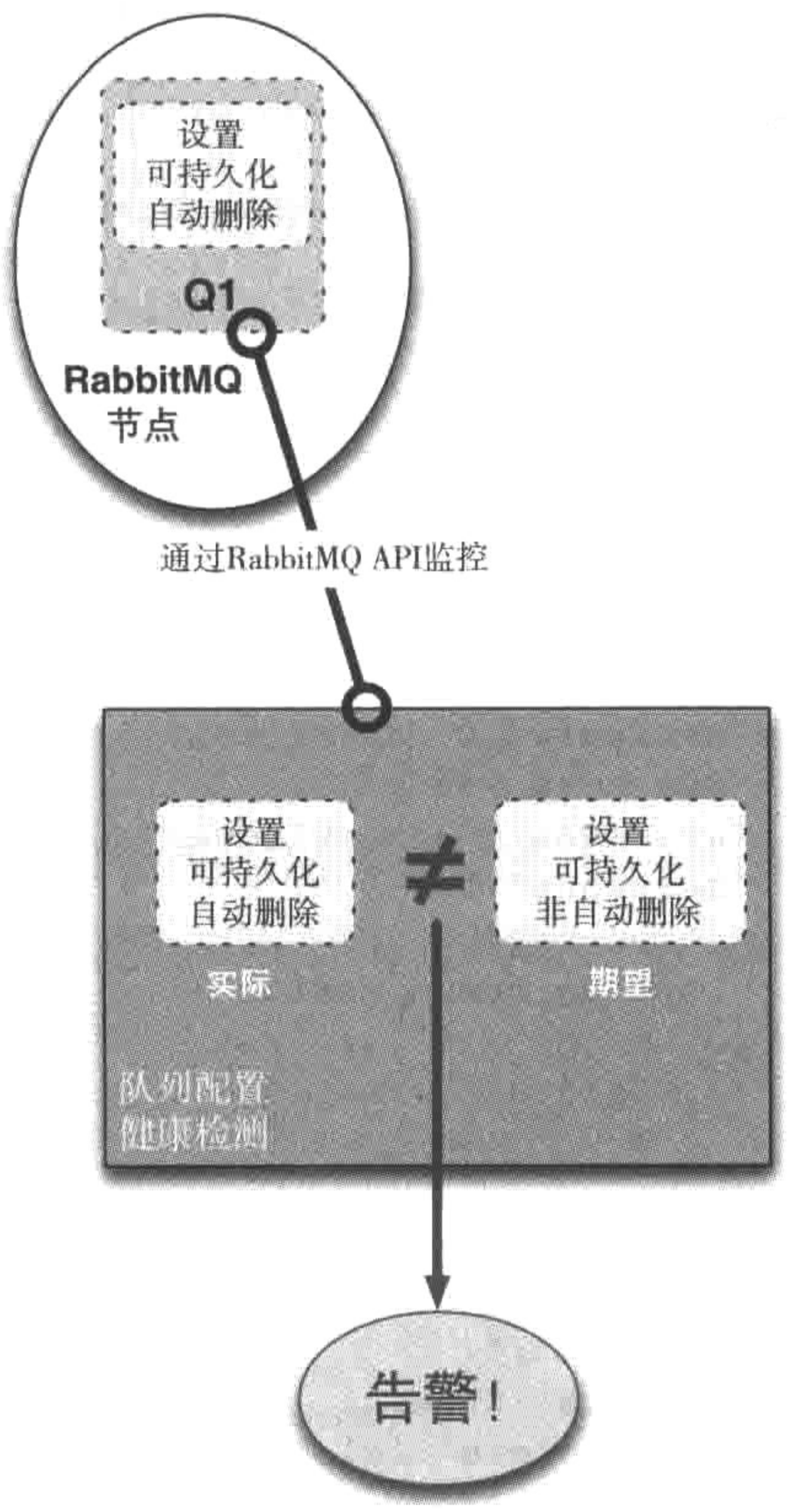

图10.1 队列配置健康检测

在Rabbit Management插件和API以前，我们很难对队列（或者交换器）的配置进行监控。唯一你可以验证队列配置的方式是，尝试用对应的参数来重新声明队列，并且相信RabbitMQ会在声明的配置与现存队列的配置不同时加以拒绝。该方法的最大问题在于，由于是通过声明队列的方式来检测故障，所以会主动更改消息通信结构。换句话说，如果健康检测有bug的话，它可能就会碰巧更改了队列的配置。幸运的是，通过使用现成的RabbitMQ API，你有了更好的解决方案。

Rabbit Management API提供了一个方法允许你查看任何vhost上的任何队列：`/api/queues/<vhost>/<queue>`。你不仅可以查看配置详情，也可以查看队列的数据统计，例如队列消耗的内存，或者队列的平均消息通信吞吐量。在下列清单中，让我们使用`curl`来快速查看在“/”vhost（注意“/”在URL中会被编码为`%2F`）上样本队列`my_queue`的配置和状态。

清单 10.4 /api/queues/<vhost>/<queue> curl 输出

```
$ curl -i -u guest:guest http://localhost:55672/api/queues/%2F/my_queue
HTTP/1.1 200 OK
Server: MochiWeb/1.1 WebMachine/1.7 (participate in the frantic)
Date: Tue, 16 Aug 2011 23:25:14 GMT
Content-Type: application/json
Content-Length: 670
Cache-Control: no-cache

{
    "memory":8400,                                    ❶ 队列内存使用
    "idle_since":"2011-8-16 17:24:46",
    "exclusive_consumer_pid":"",
    "exclusive_consumer_tag":"",
    "messages_ready":0,                               ❷ 消息计数
    "messages_unacknowledged":0,
    "messages":0,
    "consumers":0,
    "backing_queue_status":
    {
        "q1":0,
        ...
    },
    ...
    "name":"my_queue",
    "vhost":"/",
    "durable":true,                                   ❸ 队列配置
    "auto_delete":false,
    "arguments":
    {
    },
    "node":"rabbit@Phantome"
}
```

为了简洁起见去掉了部分数据统计（“…”部分），但是你仍然可以得到队列的很多信息。举例来说，你可以看到 my_queue 消耗了 8400 字节内存❶，并且队列中没有等待中的消息❷。你感兴趣的队列配置元素在 JSON 格式输出的底部❸。在这里你可以看到 my_queue 在创建时将 durable 设置为了 true，同时 auto_delete 设置为了 false（arguments 数组包含了队列可选的配置参数列表）。利用这些配置信息，新的健康检测程序可以通过 API 方法的输出来轻松监控 durable 和 auto_delete 参数，并在变更发生时通知你。

就像之前编写的健康检测程序那样，你会使用命令行参数来让检测程序知道需要检测哪台 API 服务器。但现在你需要一些额外的参数。除了通常的这几个参数：服务器、端口、vhost、用户名和密码之外，你还需要知道：

- 队列的名称，以便监控其配置
- 该队列是否将 durable 和 / 或 autodelete 选项打开

由于 API 调用会返回 durable 和 auto_delete 的 JSON 编码的布尔类型，因此你在命令行上输入期望的 true 或 false 值，并以 JSON 格式编码：

```
server, port = sys.argv[1].split(":")
vhost = sys.argv[2]
username = sys.argv[3]
password = sys.argv[4]
queue_name = sys.argv[5]
auto_delete = json.loads(sys.argv[6].lower())
durable = json.loads(sys.argv[7].lower())
```

你用 JSON 格式来解码参数并将它们分别与 API 方法输出的参数进行一一比较。这里生成 API 调用的 HTTP 请求与 API ping 健康检测程序代码非常相似。唯一改动的地方是请求路径指向了队列检测方法：

```
conn = httplib.HTTPConnection(server, port)

path = "/api/queues/%s/%s" % (urllib.quote(vhost, safe=""),
                              urllib.quote(queue_name))
method = "GET"                                                    发送API请求

credentials = base64.b64encode("%s:%s" % (username, password))
try:
    conn.request(method, path, "",
                 {"Content-Type" : "application/json",
                  "Authorization" : "Basic " + credentials})
except socket.error:
    print "UNKNOWN: Could not connect to %s:%s" % (server, port)
    exit(EXIT_UNKNOWN)

response = conn.getresponse()
```

也许你注意到了这段 API ping 健康检测中 HTTP 连接代码的另一处更改：如果健康检测无法连接到 API 服务器的话，它会返回 EXIT_UNKNOWN。你使用 API ping 健康检测程序来验证 RabbitMQ 服务器的可用性，因此 API 服务器变得不可用就意味着健康检测失败了，应该返回 EXIT_CRITICAL。但是这个新的健康检测的任务并不是监控 Rabbit 服务器的可用性，而是监控队列的配置。因此，如果无法连接到 API 服务器的话，你应该返回 EXIT_UNKNOWN，因为此时健康检测无法识别队列的配置。在健康检测程序进入未知状态时你是否能收到监控系统的告警通知，取决于你如何配置监控系统（Nagios）来处理 UNKNOWN 状态。通过返回 UNKNOWN 而非 CRITICAL，你可以提供更精确的响应，同时给予监控系统管理员一定的灵活性，

他就能以合适的方式来配置系统处理 UNKNOWN 状态。

与之前那些检测程序的区别在于，现在这个检测程序处理 API 调用返回的 HTTP 响应的方式，如下列清单所示。

清单 10.5 queue_config_check.py：API 响应处理

```
if response.status == 404:
    print "CRITICAL: Queue %s does not exist." % \
          queue_name
    exit(EXIT_CRITICAL)

elif response.status > 299:
    print "UNKNOWN: Unexpected API error: %s" % \
          response.read()
    exit(EXIT_UNKNOWN)

response = json.loads(response.read())

if response["auto_delete"] != auto_delete:
    print "WARN: Queue '%s' - auto_delete flag is NOT %s." % \
          (queue_name, auto_delete)
    exit(EXIT_WARNING)

if response["durable"] != durable:
    print "WARN: Queue '%s' - durable flag is NOT %s." % \
          (queue_name, durable)
    exit(EXIT_WARNING)

print "OK: Queue %s configured correctly." % queue_name
exit(EXIT_OK)
```

❶ 队列不存在，返回critical代码

❷ 未知的API错误，返回unknown代码

❸ 解析API响应

❹ 队列的auto_delete标记不正确，返回warning代码

❺ 队列的durable标记不正确，返回warning代码

❻ 队列存在，并且各个标记都正确，返回OK代码

API ping 健康检测要么成功要么失败，故障代码之间没有区别。但是队列检测 API 方法在失败时通过 HTTP 状态码提供了更多信息。如果 HTTP 状态码是 404，你就知道尝试验证的队列并不存在❶。由于这与你想要验证的队列配置相冲突，所以当看到 404 时，需要将退出代码设置为 EXIT_CRITICAL，并输出一个可读的消息来表示由于队列不存在，检测失败。对于任何其他大于 299 的 HTTP 状态码，你将退出代码设置为 EXIT_UNKNOWN ❷，并输出可读的消息来表示意外的 API 错误，并包含 HTTP 响应的正文。在这里，使用 EXIT_UNKNOWN 作为健康检测退出代码是合情合理的，因为曾经就是用 unknown 状态来表示意外的错误阻止了对队列配置的检测。

在剩余代码中，你将 HTTP 响应使用 JSON 解码❸，并将 auto_delete ❹和 durable ❺参数与命令行提供的参数进行比较。如果 auto_delete 和 durable 中

的任何一个与命令行参数的值不匹配的话，你将退出代码设置为 EXIT_WARNING，并提供一条描述性的可读消息。现在你也许会好奇为什么在配置参数验证失败的情况下要将健康检测状态设置为 warning 而不是 critical。老实说，这取决于你的策略和环境。如果你更关心队列是否存在，同时相对地并不在意队列实际配置的话，用 warning 状态码来代表 durable 或 auto_delete 不匹配就足够了。但是如果在电源故障发生时队列是非持久化的这一点对你来说是致命的话，你当然想要使用退出代码 EXIT_CRITICAL 了。

如果队列的所有配置都正确的话，那么就正常退出❻。通过该健康检测程序，你可以在监控系统中设置多个实例来监控应用程序中所有至关重要的队列配置。同样地，如果你需要确保某个交换器是否以特定方式进行配置的话，则可以重写检测程序，使用 /api/exchanges/<vhost>/<exchange> API 来监控交换器配置参数。与本书中大多数尝试一样，你专注于监控 Rabbit 服务器的方方面面，不论是在独立服务器上还是在集群上都能适用。如果运行的是一个集群的话，那么知道以下两点对你来说同样重要：独立节点何时消失或者节点内部数据统计超过了期望值。那么让我们来看看，为了编写程序来监控 RabbitMQ 集群健康状态需要做哪些工作。

10.1.5 监控集群状态

你也许会好奇为监控 RabbitMQ 集群来创建另一个健康检测到底有何意义？毕竟，难道不能使用之前编写的 AMQP 或者基于 API ping 健康检测来监控集群中的所有节点吗？是的，你可以这样做，并且集群之中的任何节点发生故障的时候，你都能立即知道——无须为集群做特别的健康检测。但是请想象这样的场景：你加载了一台替换服务器来替代由于硬盘问题而发生故障的节点。这台新服务器和故障服务器在各个方面，包括 IP 地址和 Erlang 节点名称完全一致。一旦加载完成，你将从监控系统中收到一个告警通知。针对该节点的 AMQP ping 健康检测告诉你 RabbitMQ 又重新启动并运行了。棒极了！你的集群又能全力运作了，你的工作也完成了！是这样吗？ 1 小时之内你开始收到客户报告说他们的遛狗订单没有出现在队列中（即便 Web 应用正在确认订单）。突然间你意识到自己犯了个错误——虽然你将服务器加载成和原来的故障 RabbitMQ 节点完全一致，但是你忘记将它加入到集群中去了！这台服务器就像是独立的 Rabbit 服务器那样，所以任何发往该服务器的

订单对连接在集群剩余节点上的应用程序来说都不可见。而一直以来，AMQP ping健康检测报告替换节点一切正常。当你使用 RabbitMQ 集群来为应用程序提供支持的时候，光是简单地确保所有节点运行正常并接收 AMQP 命令是不够的。你需要确保它们能像一个独立单元那样共同协作，同时每一个节点都加入到集群当中。需要为兼容 RabbitMQ 集群创建特别健康检测的另一个原因是：当集群节点在运行时达到了最大内存限制时，你需要能主动地获取告警通知。

有许多原因会致使 RabbitMQ 使用太多的内存，并达到 Rabbit 配置的最大内存上限。以下是最常见的几种原因：

- 应用程序有缺陷，消费消息之后忘记向 RabbitMQ 发回确认消息。这在高容量环境下，会导致成千上万条的消息堆积并耗尽 Rabbit 的内存。
- 应用程序使用 RabbitMQ 将大型数据（譬如图像）路由到处理节点。用不了多少张 100MB 大小的图像就可以将只有 8GB 内存大小的服务器内存耗尽。
- 使用了最新 RabbitMQ 版本中的新功能，但是该功能有个 bug 会导致缓慢的内存泄漏。

不管什么原因，一旦 RabbitMQ 耗尽了内存，坏事情就开始发生——譬如，RabbitMQ 开始完全没有响应或者干脆崩溃了。当 RabbitMQ 遇到内存耗尽情况时，会尝试使用磁盘来存储消息，但是如果消息量特别巨大的话，磁盘也不够存储。当集群中的某个节点发生内存耗尽的情况时，你会看到断断续续的一些奇怪的行为，对你来说不合常理。举例来说，当消息发布到节点 A 上的一个 fanout 交换器时，AMQP 信道可能出现挂起，但是节点 A 看起来是正常的。实际上，该信道正处于锁定状态，这不是因为节点 A 发生了什么错误，而是因为节点 A 上有个绑定将消息投递给节点 B 上的队列，不幸的是节点 B 耗尽了内存。由于节点 A 无法连接到节点 B 以投递消息，因此在消息投递超时前，该信道会一直处于挂起的状态。如果健康检测程序能够监控集群，并在其中一个节点快要耗尽内存之前通知你，使得你能够在节点无响应之前纠正这个问题的话，那就最好不过了。

幸运的是，你无须分开健康检测来监控集群成员和成员的内存使用情况。这是因为 Rabbit HQ 的那帮聪明的家伙让你使用单个 API 调用就可以得到自己想要知道的关于集群和集群成员的一切信息：`/api/nodes`。如果你手动使用 `curl` 来查询

/api/nodes 的话，将获得由集群中每个节点字典组成的 JSON 数组：

```
$ curl -i -u guest:guest http://localhost:55672/api/nodes
HTTP/1.1 200 OK
Server: MochiWeb/1.1 WebMachine/1.7 (participate in the frantic)
Date: Thu, 18 Aug 2011 02:15:10 GMT
Content-Type: application/json
Content-Length: 4254
Cache-Control: no-cache
[
    {
        "name":"rabbit@Phantome",
        "type":"disc",
        "running":true,
        ...
        "mem_used":31537360,
        "mem_limit":1675968512,
        "mem_alarm":false,
...
```

每个节点字典包含了该节点上的数据统计和配置元素。举例来说，你可以看到节点 rabbit@Phantome 是一个 disc 类型节点，并使用了 31537360 字节（30 MB）的内存（mem_used）。如果你使用了独立 Rabbit 服务器的话，也可以通过查询 /api/nodes 来获取服务器信息；不同的是这时你只会得到单个节点字典，而不是集群中多个节点字典。在对 /api/nodes 响应内容基本了解之后，你已经准备好开始构建集群健康检测了。由于基于 /api/nodes 的健康检测会同时监控节点成员和内存使用，因此该检测程序首先需要知道检测哪些集群成员，以及 warning 状态或者 critical 状态所指示的内存级别。为了获取这些设置，你将添加三个新的参数（节点列表、内存使用 warning 阈值、内存使用 critical 阈值）到平常需要的服务器地址、端口号和凭证参数中去：

```
server, port = sys.argv[1].split(":")
username = sys.argv[2]
password = sys.argv[3]
node_list = sys.argv[4].split(",")
mem_critical = int(sys.argv[5])
mem_warning = int(sys.argv[6])
```

节点列表参数比较特殊，检测程序期望该参数是以逗号分隔的节点列表（例如 rabbit@node1,rabbit@node2）。这样的话就可以让用户将节点列表作为单一参数传入，而不是传入多个参数。在设置了检测程序的配置设定之后，你准备好连接到 API 服务器并将请求 post 到 /api/nodes，如下列清单所示。

清单 10.6 cluster_health_check.py：将请求发往 /api/nodes

```
conn = httplib.HTTPConnection(server, port)                    ← ❶ 连接到服务器

path = "/api/nodes"                                            ← ❷ 构建API路径
method = "GET"

credentials = base64.b64encode("%s:%s" % (username, password))
try:                                                           ← ❸ 发送API请求

    conn.request(method, path, "",
                 {"Content-Type" : "application/json",
                  "Authorization" : "Basic " + credentials})
except socket.error:
    print "UNKNOWN: Could not connect to %s:%s" % (server, port)
    exit(EXIT_UNKNOWN)

response = conn.getresponse()

if response.status > 299:
    print "UNKNOWN: Unexpected API error: %s" % response.read()
    exit(EXIT_UNKNOWN)
```

如同之前基于 API 的健康检测程序一样，你通过 HTTP 连接到 API 服务器❶，并通过 Base64 编码的头来发送凭证❸。唯一不同的地方在于该健康检测程序将发送请求到 /api/nodes ❷。在发送请求之后，你准备开始处理 API 服务器的响应，如下列代码清单所示。由于你需要检测节点成员看看是否有缺失，同时也要检测每个节点的内存使用情况，因此你将对包含数据和配置的节点字典数字进行两次遍历。

清单 10.7 cluster_health_check.py：处理节点列表

```
response = json.loads(response.read())                         ← ❶ 解析API响应

for node in response:
    if node["name"] in node_list and node["running"] != False:
        node_list.remove(node["name"])                         ← ❷ 集群节点缺失，返回warning

if len(node_list):
    print "WARNING: Cluster missing nodes: %s" % str(node_list)
    exit(EXIT_WARNING)

for node in response:                                          ← ❸ 节点的内存使用超过限制
    if node["mem_used"] > mem_critical:
        print "CRITICAL: Node %s memory usage is %d." % \
        (node["name"], node["mem_used"])
        exit(EXIT_CRITICAL)
    elif node["mem_used"] > mem_warning:
        print "WARNING: Node %s memory usage is %d." % \
              (node["name"], node["mem_used"])
        exit(EXIT_WARNING)

print "OK: %d nodes. All memory usage below %d." % (len(response),   ← ❹ 所有节点都存在，并且使用的内存未超过限制
                                                    mem_warning)

exit(EXIT_OK)
```

在用 JSON 解码节点字典数组为对应的 Python 变量之后❶，你尝试将字典中的每个节点 name 元素❷与从命令行传入的期望成员的节点相匹配。随着迭代节点字典数组，如果字典中的 name 元素和期望的节点名称相匹配的话（并且该节点处于 running），那么就从期望节点列表中移除该名称。结果是如果所有期望的节点都存在的话，那么在结束遍历节点字典时，期望节点列表就会为空。另一方面，如果期望节点列表中仍然有条目的话，那么根据 /api/nodes，你就知道这些节点不存在了，或者没有运行。如果检测程序确定集群中缺失了节点，那么你将退出代码设置为 EXIT_WARNING 并退出，不再进行其他分析了。将状态设置为 warning 的原因在于缺失的节点虽然会使得集群在性能上有所损失，但是却不会使得集群无法正常工作。如果你觉得缺失节点非常严重的话，那么就将退出代码设置为 EXIT_CRITICAL。

一旦你确认了所有期望节点都存在于集群之中的话，然后你就再一次迭代每个节点字典，求出每个节点使用的内存大小❸。如果任何节点字典中的 mem_used 元素大于通过命令行设置的 critical 内存使用阈值的话，那么就将状态代码设置为 EXIT_CRITICAL 并退出，同时输出一段可读信息表示受影响的节点当前的内存使用数据（以字节表示）。如果节点的 mem_used 元素没有超过 critical 阈值，但是超过了内存使用的 warning 阈值的话，你将以 EXIT_WARNING 状态代码退出，并输出受影响节点的当前内存使用情况。最后，如果所有期望的集群成员都存在，并且没有一个超过了 warning 或者 critical 内存使用阈值的话，就以 EXIT_OK 状态代码退出❹，并输出所有节点内存使用均低于 warning 阈值数值。让我们这就来运行健康检测程序，看看会发生什么吧：

```
$ python cluster_health_check.py localhost:55672 guest guest \
  rabbit@Phantome,rabbit2@Phantome 34000000 33000000
OK: 2 nodes. All memory usage below 33000000.
$ echo $?
0
```

棒极了，你的健康检测程序确认了 rabbit@Phantome 和 rabbit2@Phantome 节点是集群中的成员，并且两者的内存使用情况均低于 critical（34 MB）和 warning（33 MB）阈值。不过，让我们来看看你的检测程序如何处理其中一个集群节点消失的情况：

```
$ rabbitmqctl -n rabbit2@Phantome stop_app
Stopping node rabbit2@Phantome ...
...done.
```

```
$ python cluster_health_check.py localhost:55672 guest guest \
  rabbit@Phantome,rabbit2@Phantome 34000000 33000000
WARNING: Cluster missing nodes: ['rabbit2@Phantome']
$ echo $?
1
```

到目前为止一切顺利；你的健康检测程序不仅检测到了集群中有节点缺失，同时能告诉你是哪个节点缺失了。最后，你需要确定检测程序能够正确地对节点进行warning 和 critical 内存阈值检测：

```
$ python cluster_health_check.py localhost:55672 guest guest \
  rabbit@Phantome,rabbit2@Phantome 32000000 31000000
WARNING: Node rabbit2@Phantome memory usage is 31785552.
$ echo $?
1
$ python cluster_health_check.py localhost:55672 guest guest \
  rabbit@Phantome,rabbit2@Phantome 31000000 30000000
CRITICAL: Node rabbit2@Phantome memory usage is 31834792.
$ echo $?
2
```

通过将 critical/warning 内存阈值在 32 MB/31 MB 和 31 MB/30 MB 两者之间进行切换，你可以看到当 RabbitMQ 使用了超出限定的内存时，你的新检测程序能够正确检测并报告 。现在你拥有一套完整的面向 Rabbit 的健康检测。它不仅能在单台 RabbitMQ 服务器不可用的时候告知你，同时也能在 Rabbit 集群缺失节点或者任何成员节点快要耗尽内存的时候及时通知你。但是如果问题不在于 Rabbit 是否正常运作，而是在于你的应用程序是否能像原本那样正确地消费消息呢？集群健康检测可以让你知道内存是否耗尽，但是如果问题在于你的遛狗应用未能正确消费订单队列中的订单的话，在内存耗尽成为问题之前就能够知道，这不是很好吗？如果你的消息容量（或者大小）很低，则监控特定队列中有多少条消息显得更为重要，因为这时消息会慢慢累计起来而不会引起注意，直到客户来抱怨他们的订单没有被处理。记住这些，现在该来看看如何构造健康检测程序来检测你的消费者是否停止消费了。

10.2 确保消费者正常工作

到目前为止，我们主要关心的是确保 RabbitMQ 服务器正常运行，能够路由消息，并且正确地集群。但是我们没有讨论过使用消息通信会造成一个缺点：监控消费者会变得更加困难。让我们拿遛狗网站作为示例。该网站的关键部分是幕后持续

运行的守护进程，用来处理由面向用户的 Web 应用记录的订单。如果没有使用消息通信的话，你很有可能将订单处理守护进程设计为通过 HTTP 通信的服务器应用程序。从客户收集订单信息之后，Web 应用将会连接到订单处理应用监听的服务器端口。一旦连上之后，就会开始传送订单。我们曾经讨论过，该方法最大的缺陷在于，直到订单处理应用确认接管订单之后，通常是在订单处理完之后，Web 应用才能返回，然后继续接收下一个订单。前端 Web 应用和后端订单处理器两者之间的解耦带来的好处要归功于消息通信的使用。但随之而来的问题是你如何监控订单处理器是否正常运作。当对它们进行监控时，这些接收 TCP 连接的服务器类型应用易于监控。一旦你将该应用转换为使用消息通信的话，就无法再连接到服务器应用的监听端口来确认其是否启动并能够正常处理订单了。因为根本没有可以连接的端口！

别怕！即便订单处理应用使用的是消息通信，你仍然可以对其进行监控。你要做的只是换一种思维模式。如果消费者无法消费消息和处理消息的话，那么随之而来的副作用就是消息会在供应给消费者的那个队列上堆积起来。你也许还记得在本书前几章，我们曾鼓励在编写消费者时，尽量等到处理完消息之后才对收到的消息进行确认。我们之所以鼓励这样的处理方法的原因之一（除了确保消息不会丢失之外）是，如果消费者在处理消息的时候持续处于崩溃状态，那么这些消息会在队列中累积起来，而健康检测程序就检测到并触发告警。因此，你可能已经猜到了，监控消费者是否正确运作的方式就是通过监控队列的消息总数，并在总数超过设定的 warning 或者 critical 阈值时触发告警。

如同你之前编写的 ping 健康检测一样，你可以用两种方式来监控队列消息总数：

（1）使用 AMQP 的 `queue_declare()` 命令，设置 `passive=True` 参数来重新声明一个已存在的队列。当你在 AMQP 中声明一个队列时，如果将 `passive` 设置为 `True` 的话，那么该命令返回的结果中将包含队列消息的总数。

（2）利用我们的老朋友 Rabbit Management API 来从队列上拉取数据统计，其中就有队列当前的消息总数。

在学习了如何构建队列消息总数检测（以及每个方法的优势）的每个版本之后，我们将要看到的是如何分析消息通信量来为应用程序使用的每个队列找出合适的 warning 和 critical 阈值。事不宜迟，让我们动手使用 AMQP 来监控队列中的消息总数吧。

10.2.1 通过 AMQP 监控队列等级

你可能会问自己一个问题："为什么我们要用 AMQP 呢？使用 API 的话不是能提供更多细节吗？"简而言之，的确如此。API 提供一揽子的消息数据，包括等待消费的消息总数，以及已经投递给消费者但尚未被确认的消息总数。使用 AMQP 来监控消息总数的话，你只能得到队列中消息的聚合数据，对未消费以及未确认消息没有做区分。但是如果你使用的是不支持 Management 插件的 RabbitMQ 早期版本，或者由于技术 / 安全的原因你无法安装 Management 插件的话，那么使用 AMQP 来监控消息总数就是你的唯一出路了。要么不去监控队列，要么使用基于 AMQP 的检测程序来监控，二者选其一，如果应用程序可靠性对你来说至关重要的话，就没什么好犹豫的了。

和之前编写的基于 AMQP ping 健康检测相同的是，AMQP 消息总数检测程序需要在命令行上获取服务器地址、端口、虚拟主机和凭证作为参数。它需要额外的参数，以便清楚需要监控哪个队列，以及基于怎样的阈值来划分 warning 或者 critical 状态：

```
server, port = sys.argv[1].split(":")
vhost = sys.argv[2]
username = sys.argv[3]
password = sys.argv[4]
queue_name = sys.argv[5]
max_critical = int(sys.argv[6])
max_warn = int(sys.argv[7])
```

新的健康检测程序也会建立到 Rabbit 的连接，和 AMQP ping 检测程序完全一样。在连接上之后，它会为指定的队列发送 channel.queue_declare 来获取当前的消息总数：

```
creds_broker = pika.PlainCredentials(username, password)
conn_params = pika.ConnectionParameters(server,
                                        virtual_host = vhost,
                                        credentials = creds_broker)
try:
    conn_broker = pika.BlockingConnection(conn_params)
    channel = conn_broker.channel()
except socket.timeout:
    print "Unknown: Could not connect to %s:%s!" % (server, port)
    exit(EXIT_UNKNOWN)

try:
    response = channel.queue_declare(queue=queue_name,
                                     passive=True)
```

```
except pika.exceptions.AMQPChannelError:
    print "CRITICAL: Queue %s does not exist." % queue_name
    exit(EXIT_CRITICAL)
```

被监控队列的当前消息总数包含在 response 对象内。但在我们访问消息总数之前，值得注意的是传递给 queue_declare 的 passive=True 参数。passive 参数告诉 RabbitMQ 你并不想真正声明队列，而是想知道队列是否存在。通过将 passive 设置为 True，如果队列不存在的话，queue_declare 会引发异常；否则，返回队列当前的消息总数。确认在方法调用时传递了 passive 参数这一点至关重要。这不仅是返回队列消息总数的唯一方法；同时也是因为，如果没有传递该参数的话，检测程序会真实地尝试声明该队列。在被动（passive）queue_declare 操作完成之后，你将获得包含消息总数的 response 对象。为了获取消息总数，你引用了 .method.message_count 属性，如下列清单所示。

清单 10.8 amqp_queue_count_check.py：验证队列消息总数

```
if response.method.message_count >= max_critical:
    print "CRITICAL: Queue %s message count: %d" % \
          (queue_name, response.method.message_count)
    exit(EXIT_CRITICAL)

if response.method.message_count >= max_warn:
    print "WARN: Queue %s message count: %d" % \

          (queue_name, response.method.message_count)
    exit(EXIT_WARNING)

print "OK: Queue %s message count: %d" % \
      (queue_name, response.method.message_count)
exit(EXIT_OK)
```

❶ 消息总数超过了critical限制

❷ 消息总数超过了warning限制

❸ 连接OK，返回OK

由于属性 message_count 是整型，因此很容易将它与检测程序的消息总数阈值进行比较。如果 message_count 大于 critical 阈值的话❶，那么就以 EXIT_CRITICAL 状态码退出；如果它低于 critical 阈值但是高于 warning 阈值的话，那么就以 EXIT_WARNING 状态码退出❷。另一方面，如果 message_count 没有超过 critical 和 warning 阈值的话，那么设置状态码为 EXIT_OK ❸并退出。除了设置 AMQP 连接 Rabbit 的代码外，该健康检测程序其实很简单。

为了测试这段检测代码，首先你将使用在第 8 章介绍的 Rabbit Management Web UI 来创建名为 my_queue 的队列，如图 10.2 所示。

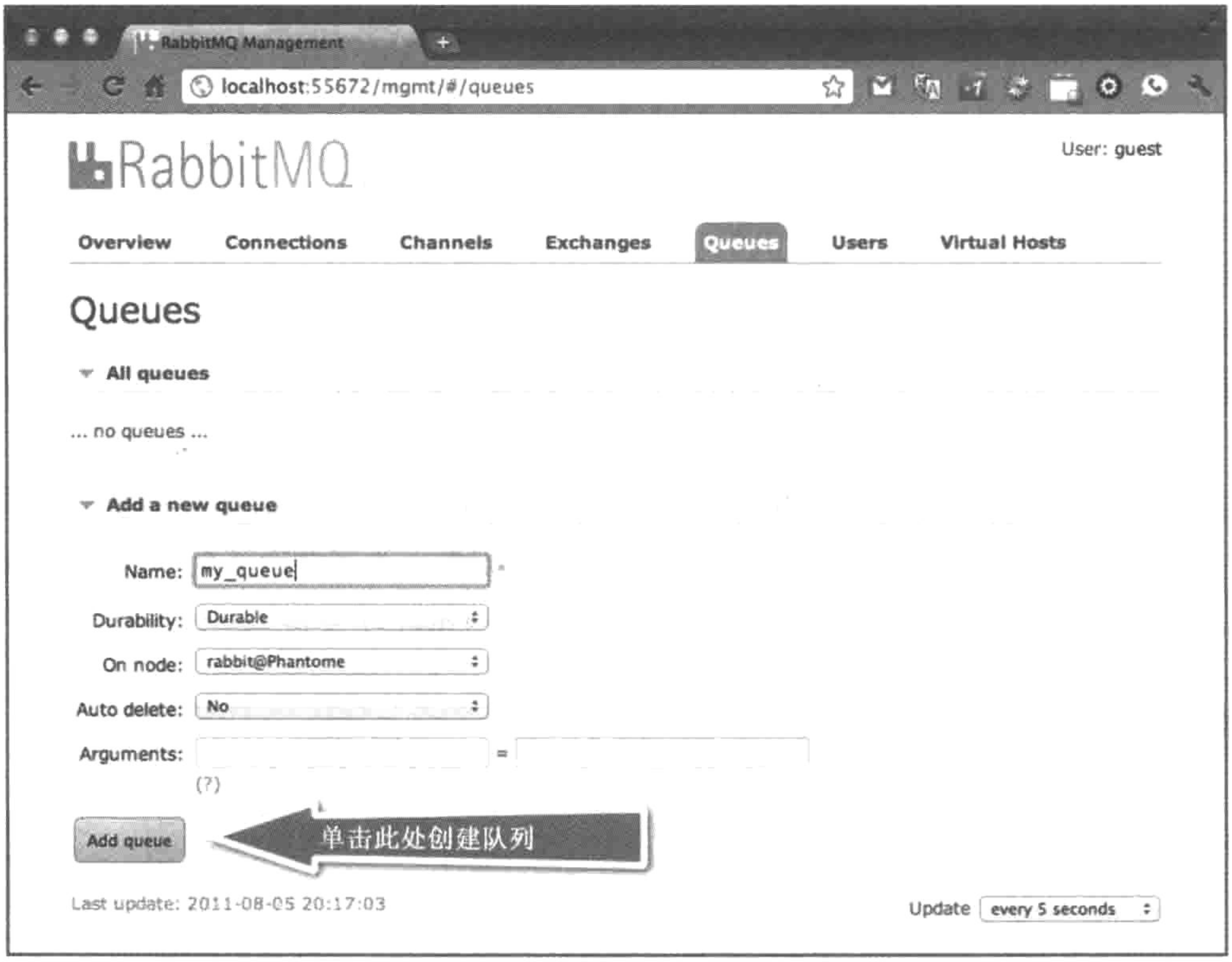

图 10.2 创建 **my_queue**

创建完队列之后，下一步你将单击 Management Web UI 上的 Exhange 选项卡，然后在结果清单页上单击（AMQP default）。使用（AMQP default）交换器的原因在于测试消息使用 my_queue 作为路由键，并发布到该交换器的话，消息会自动路由到 my_queue，而不用显式地设置绑定。在 Management Web UI 上的（AMQP default）页面使用 Publish message 按钮，发布一条带有 Any payload will do 内容、路由键为 my_queue 的消息。在收到 Message published 通知之后，再一次单击 Publish message 按钮将消息的第二份拷贝发送到 my_queue 中，如图 10.3 所示。

现在你应该在 my_queue 队列上有两条等待的消息，那么让我们运行健康检测程序来看看是否能观测到消息：

```
$ python amqp_queue_count_check.py localhost:5672 / guest \
  guest my_queue 4 3
OK: Queue my_queue message count: 2
$ echo $?
0
```

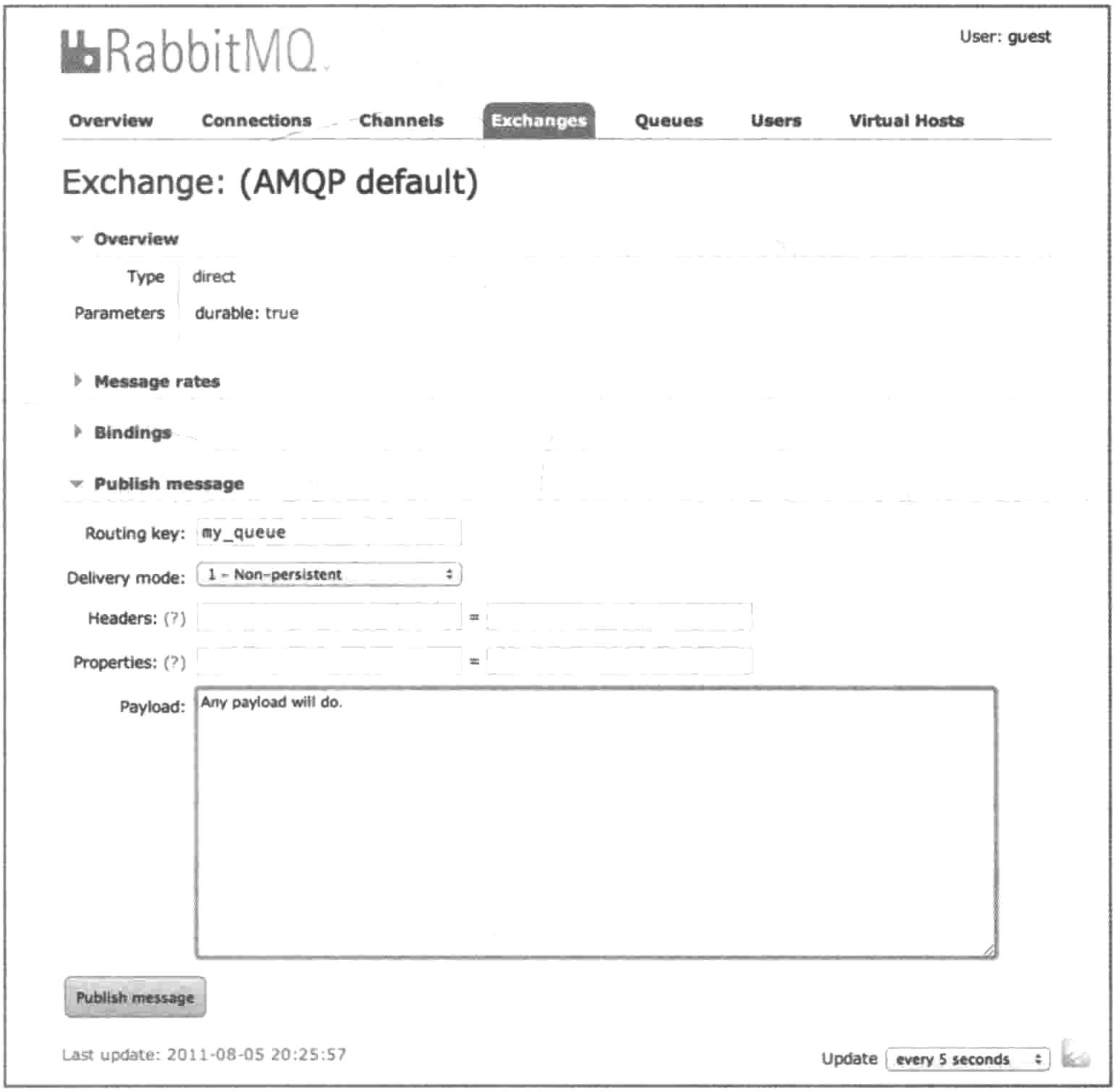

图 10.3 将测试消息发布到 my_queue

你告诉健康检测程序判定四条（含）以上的消息为 critical 状态，同时三条（含）以上为 warning 状态。因此，该检测程序返回 OK 状态，因为队列 my_queue 中只有两条消息，一切正常。让我们更改 critical 和 warning 阈值分别为三条和两条消息，并重新运行：

```
$ python amqp_queue_count_check.py localhost:5672 / guest \
  guest my_queue 3 2
WARN: Queue my_queue message count: 2
$ echo $?
1
```

现在，健康检测程序理所当然地返回 warning 状态，因为两条消息达到了队列消息总数的 warning 阈值，但是尚未超过三条消息的 critical 阈值。如果再次减少 critical 和 warning 阈值为两条和一条消息的话，该检测程序应该返回 critical 状态：

```
$ python amqp_queue_count_check.py localhost:5672 / guest \
  guest my_queue 2 1
CRITICAL: Queue my_queue message count: 2
$ echo $?
2
```

只不过 50 行代码，你就拥有了监控任何队列的消息总数的方法，并且能够在队列消息总数增长到危险级别之前就未雨绸缪。有时，消息总数达到了惊人的高度，不是因为消费者崩溃了，而是可能由于消费者在成功处理消息之后没有对消息进行确认。如果能够知道过高的消息总数是由于未消费（崩溃了的消费者）还是未确认（有 bug 的消费者）的话，岂不是帮助很大？虽然基于 AMQP 的检测无法给予你这类信息，但是 Rabbit API 却可以。让我们看看你如何创建一个更强大的消息总数检测程序，通过使用 Rabbit Management API 来解决上述问题。

10.2.2 使用 REST API 来监控队列级别

当你通过 API 来构建健康检测程序监控队列配置的时候，你可能会记得 /api/queues/<vhost>/<queue_name> 返回的配置信息中散落着各种数据统计信息。在其中就有队列当前消息总数：

```
...
"messages_ready":0,
"messages_unacknowledged":0,
"messages":0,
...
```

不过由 /api/queues/<vhost>/<queue_name> 产生的输出并不仅仅告诉你队列上的消息总数（messages），它同时还将聚合消息计数分解为未消费消息（messages_ready）和未确认消息（messages_unacknowledged）。这些都是很有用的信息。举例来说，假设由于在 The Dog Whisperer 上投放了广告之后，你的遛狗订单应用几乎每天都有一波高峰。在这期间，订单队列上的消息增长了 10 倍。如果使用 AMQP 健康检测程序的话，这势必会超过 critical 阈值，同时每天都会触发误报。另一方面，如果使用 API 检测消息总数的话，你会看到这里 10 倍的增长仅限于messages_ready（未消费消息）。通过使用 API 创建新的消息计数检测程序，

你能够为未消费消息设置更高的阈值，同时为未确认的消息维持较低的阈值。这样做你就可以降低未消费消息总数上的误报次数，同时能够在未确认消息总数高企时快速收到通知。

与到目前为止所有的健康检测一样，首先你需要获取 RabbitMQ 连接和从命令行传入的认证信息。除此之外，还需要获取监控的队列名称，以及未消费（等待）和未确认消息计数这两者的 critical 和 warning 阈值：

```
server, port = sys.argv[1].split(":")
vhost = sys.argv[2]
username = sys.argv[3]
password = sys.argv[4]
queue_name = sys.argv[5]
max_unack_critical = int(sys.argv[6])
max_unack_warn = int(sys.argv[7])
max_ready_critical = int(sys.argv[8]
max_ready_warn = int(sys.argv[9])
```

在设置了健康检测之后，你准备好连接到 API 服务器。与之前基于 API 的健康检测一样，你会使用相同的连接代码。不过这次你将连接到 /api/queues/<vhost>/<queue_name>：

```
conn = httplib.HTTPConnection(server, port)
path = "/api/queues/%s/%s" % (urllib.quote(vhost, safe=""),
                              queue_name)
method = "GET"
credentials = base64.b64encode("%s:%s" % (username, password))
try:
    conn.request(method, path, "",
                 {"Content-Type" : "application/json",
                  "Authorization" : "Basic " + credentials})
except socket.error:
    print "UNKNOWN: Could not connect to %s:%s" % (server, port)
    exit(EXIT_UNKNOWN)
response = conn.getresponse()
if response.status > 299:
    print "UNKNOWN: Unexpected API error: %s" % response.read()
    exit(EXIT_UNKNOWN)
```

假设 API 连接成功并且请求没有导致任何 HTTP 错误，你现在可以分析响应来确定队列当前的消息计数级别。在对响应进行 JSON 解码后，你就可以从响应字典中获取 messages_unacknowledged 和 messages_ready：

```
resp_payload = json.loads(response.read())
msg_cnt_unack = resp_payload["messages_unacknowledged"]
```

```
msg_cnt_ready = resp_payload["messages_ready"]
msg_cnt_total = resp_payload["messages"]
```

最后，在获得未消费（等待）和未确认消息计数之后，你可以将它们与提供的阈值进行比较，如下列清单所示。

清单 10.9 api_queue_count_check.py：检查消息计数

```
if msg_cnt_unack >= max_unack_critical:
    print "CRITICAL: %s - %d unack'd messages." % (queue_name,
                                                  msg_cnt_unack)
    exit(EXIT_CRITICAL)
elif msg_cnt_unack >= max_unack_warn:
    print "WARN: %s - %d unack'd messages." % (queue_name,
                                              msg_cnt_unack)
    exit(EXIT_WARNING)

if msg_cnt_ready >= max_ready_critical:
    print "CRITICAL: %s - %d unconsumed messages." % (queue_name,
                                                     msg_cnt_ready)
    exit(EXIT_CRITICAL)
elif msg_cnt_ready >= max_ready_warn:
    print "WARN: %s - %d unconsumed messages." % (queue_name,
                                                 msg_cnt_ready)
    exit(EXIT_WARNING)

print "OK: %s - %d in-flight messages. %dB used memory." % \
      (queue_name, msg_cnt_total, resp_payload["memory"])
exit(EXIT_OK)
```

❶ 消费了但是未确认的消息计数超过了阈值

❷ 等待消费的消息计数超过了阈值

❸ 消息计数低于阈值，返回OK

首先要检测的是未确认消息计数是否超过了 critical 或者 warning 级别❶。如果是的话，则将退出代码设置为 EXIT_CRITICAL 或 EXIT_WARNING（取决于超过了哪个阈值），并输出队列中当前未确认消息的总数。假设未确认消息计数未超过任何阈值，下一步将分析未消费（等待）消息计数❷。同样地，如果未消费消息计数超过了 critical 或者 warning 阈值的话，将退出代码设置为 EXIT_CRITICAL 或 EXIT_WARNING，然后输出未消费消息的总数。最后，如果未消费消息和未确认消息计数均未超过对应的 critical 或者 warning 阈值的话，那么将状态退出码设置为 EXIT_OK，以便让 Nagios 知道一切正常❸，同时输出队列中的消息总数（未消费 + 未确认）和队列当前消耗的内存量。

虽然测试未确认消息阈值较为复杂，但是你应该可以使用队列 my_queue 中的消息（之前测试基于 AMQP 的消息总数检测时创建）来验证这个新的基于 API 的检测程序。如果你分别将未确认的 critical/warning 阈值设置为两条和一条消息的话，

API 消息计数检测应该能正确地检测到未消费（等待）消息超过了阈值：

```
$ python api_queue_count_check.py localhost:55672 / guest \
  guest my_queue 2 1 2 1
CRITICAL: my_queue - 2 unconsumed messages.
$ echo $?
2
```

如果你分别将 critical/warning 未消费阈值提升到四条和三条消息的话，检测程序应该认为队列中的未消费消息计数是正常的：

```
$ python api_queue_count_check.py localhost:55672 / guest \
  guest my_queue 4 3 4 3
OK: my_queue - 2 in-flight messages. 9800B used memory.
$ echo $?
0
```

检测程序报告了队列中共有两条消息，同时该队列消耗了 9800 字节的内存（实际内存使用不尽相同）。现在你手里的健康检测程序能够区分到底是队列中的未确认消息还是未消费消息超过了阈值。这能帮助你更快地确认消息总数表明的是由负载增加导致的，还是由应用程序中的缺陷导致的。剩下的问题是：如何为队列确定消息总数的基准线，以便为健康检测程序设置 critical 和 warning 阈值？

10.2.3 建立队列的消息计数基准经验法则

有许多方法来确认队列消息总数的 critical 和 warning 阈值。如果你正将现存的应用转换为使用消息通信架构的话，其中的日志文件可以提供合理而又准确的信息来源。举例来说，如果应用程序处理信用卡订单的话，你可能已经将每笔订单的时间戳记录到了数据库当中。通过使用这些日志观测 10 秒时间间隔内能够处理多少笔订单，就是很好的经验法则。该数值通常来讲可以作为你的 warning 阈值，这是因为健康检测程序在运行的时候就是对队列消息总数取了一次快照，而这个快照通常代表的是 1 秒甚至是更短的时间段。如果队列中的消息总数在任何给定的时间上超过了 10 秒内消息 / 订单处理总数的话，那么就有问题了。同理，将 critical 阈值设置为 20 秒内处理的订单 / 消息总数。记住，该方法是一种估算，你需要监控真实的队列级别来确认这些阈值对你的环境来说是否正确。

确定 warning 和 critical 消息总数阈值的最好方法是使用 Cacti 或 Graphite 这样的图形监控系统来监控队列。通过对消息总数检测程序进行修改来与这些系统一同

工作，它们能够通过定期采样数据来图形化真实的队列消息总数。由此产生的图形结果将向你展示近乎精确的真实平均未消费和未确认消息总数。在掌握了这些数据之后，增加 20% 就可以得到 warning 阈值，而增加 100% 就可以得到 critical 阈值。通常来讲，任何高于正常情况 20%~90% 的情况都属于正常波动。但是高于正常情况 100% 或者更高的话，那绝对值得你去探个究竟，因为肯定已经出了差错。

10.3 总结

一开始，你也许已经有了一个健壮的 RabbitMQ 架构，但却无法对它进行监控或者确保它真正可靠。现在你已经构建了健康检测程序，它不仅可以查询 RabbitMQ 服务器以确保其可以处理 AMQP 命令；又能监控实际的消息总数级别，用来确定从队列上消费的消费者的健康情况。此外，你也能对队列的配置进行监控，以确保人为错误不会将你的队列从持久化更改为非持久化，从而避免下次服务器发生故障时演变为灾难。监控 RabbitMQ 是确保服务器正常运行并为应用程序高效地提供动力的至关重要的组成部分。一旦开始监控 RabbitMQ，这些健康检测程序就能够让你观测到低效和问题，你可以通过调整 Rabbit 来进行纠正。记住这些，现在该来看看分析 RabbitMQ 性能和行为的方法，这样你就能在应用程序中将两者最大化。

11 提升性能，保障安全

本章要点

- 交换器、队列和绑定的内存占用
- 消息持久化和磁盘I/O
- RabbitMQ的SSL连接
- 设置私钥架构

在之前的几个章节中，你已经知道如何设计消息通信系统架构了。你见识过了使用不同种类的 AMQP 构造块，诸如交换器、队列和绑定来实现多种消息通信模式。根据你手头遇到的问题，你选择这些构造块的特定组合来实现解决方案。如果你需要在多台机器上分发日志的话，则可以遵循 pub-sub 模式，使用 topic 或者 fanout 交换器；如果你需要点对点通信，那么就使用 direct 交换器，等等。在本章中，我们会回顾这些设计决策的性能。你将会看到使用 direct 交换器和 topic 交换器的利弊对比；交换器、队列或者绑定规则的最小内存占用；与 fanout 交换器相比，当你拥有数以百计的绑定规则附加在 topic 交换器上的话，会如何呢？同时你可能也会有这样的疑问，消息何时写入磁盘？服务器如何处理众多的内存队列？我们的目标是提供足够的信息，以便在涉及容量规划时帮助决策。我们通过分析消息从生产者路由到消费者的路径，基于多个 AMQP 选项，来完成实现。

在我们讨论性能之后，你将看到如何保障 RabbitMQ 安装的安全性；更具体地说，如何使用 SSL 来建立到代理服务器的可信赖的连接。我们会讲到在 RabbitMQ 中启用SSL监听器的配置方面，如何生成SSL证书，以及如何使用SSL来连接代理服务器。

让我们先看看 11.1 节，在这里你会明白，从 AMQP 的角度来看，是什么影响了消息通信的投递速度。

11.1 对速度的需求

有很多因素可以影响 RabbitMQ 投递消息的速度，这依赖于硬件和软件的配置。在硬件配置方面，这些因素有网络配置、磁盘管理、处理器核心数等。在软件级别，你可以配置多个 AMQP 参数，例如消息持久化、路由算法、绑定数目，以及消息确认策略。由于前者完全取决于装机情况，因此在本章中我们就不展开了。我们会专门讨论 AMQP 细节以及 RabbitMQ 如何对它们进行设置。

让我们从回顾消息的持久化和消息确认机制是如何影响消息投递的开始。

11.1.1 消息持久化

当讨论软件性能时，所有事情都应该谨慎对待。为什么呢？因为你做的任何决定都应该顾及到整个应用场景。你做的任何决定都有利有弊。举个例子，你能加快系统日志的消息投递速度，这些消息甚至都不用持久化到磁盘上，但是对那些购物车的订单处理来说，你最好确保处理过程万无一失。因此，虽然你可以把日志信息作为非持久化消息发送出去，然后消费它们而无须确认，但是你可承担不起以这种方式来同样处理与客户的金钱交易。所以每当处理这种类型的性能调优时，你都应该再三思考所做的取舍。

当发布消息时，你需要决定丢失其中的任何消息对你来说是否可以接受。如果丢失了几千条消息当中的几条（不管什么原因）对你来说可以接受的话，那么可以将 `delivery-mode` 设置为 1（即非持久化的），然后发布它们。通常情况下，你会通过将 `delivery-mode` 设置为 2 来将消息设置成持久化的，然后再发送。但是这是有代价的，服务器需要把消息写到磁盘上。举例来说，在一台 Mac 上，RabbitMQ 可以轻轻松松以最多每秒 12 000 条的速度投递消息。如果启用消息持久化的话，那么速度会一下子掉到每秒 4000 条左右的样子。虽然这样的速率仍然很高，但是相

比之前已经大大地降低了。

你如何管理这个设定呢？你需要为每一条发往服务器的消息指定该属性。下列代码片段展示了如何用 PHP 库来创建非持久化消息：

```
$msg = new AMQPMessage("Test Message", array('delivery_mode' => 1));
```

另一个影响消息通信速度的设定是消息确认。让我们看看吧。

11.1.2 消息确认

前面我们介绍了消息发布的设定；现在该来看看在消息消费过程中你该如何配置了。其中一个可以加快消息投递的设定是 no-ack 标记，你可以在队列订阅时指明。如果设置为 true，服务器就会在消息发送给客户端后自动将其出队。如果由于某些原因连接中断了，或者你的客户端应用程序发生故障了，那么该消息就永远丢失了。

如果在订阅队列时将 no-ack 设置为 true 的话，那么你处理完消息之后就无须再发送确认消息回服务器。这样就能极大地加快消费者消费的速度。在服务器端，由于 RabbitMQ 在消息投递之后就无须再关注了，因此消息处理的过程得到了简化。以下是 Python 代码片段，它向你展示了如何从队列消费消息，并将 no-ack 设置为 true：

```
channel.basic_consume(critical_notify,
                      queue="critical",
                      no_ack=True,
                      consumer_tag="critical")
```

你将从 critical 队列消费消息，并且也用 critical 作为消费者标记。对于收到的每一条消息，回调函数 critical_notify 都会被调用。现在该看看消息路由期间发生了什么。在下一节里，我们将分别介绍 RabbitMQ 使用的几种主要路由算法。

11.1.3 路由算法和绑定规则

在本书中，我们已经讨论过了三种不同类型的交换器：direct、fanout 和 topic。每种交换器类型代表了服务器实现的特定路由算法。当需要交换器来路由一条消息时，它会根据消息的路由键以及队列与交换器之间的绑定来选择队列。由于每种交

换器类型都会区别对待消息路由键，因此队列的选择过程会因为交换器类型的不同而不同。

在服务器端，交换器和绑定作为记录条目存储在 Mnesia 中。这意味着当 RabbitMQ 匹配消息路由键时，它会尝试查找对应该路由键的绑定。Mnesia 是一个高性能数据库，它的存储是基于 Erlang 的 ETS 和 DETS 表的[1]。ETS 指的是 Erlang Term Storage（Erlang 数据存储），它将数据存储在内存中；而相应地，DETS 是一个基于磁盘的存储方案。使用基于普通 ETS 函数调用的 Mnesia 的好处是，Mnesia 可以协调在整个集群中对表的访问。举个例子，当你在一个集群节点上创建了交换器的时候，Mnesia 会负责将该信息复制到集群中的所有其他节点上；而对于添加绑定、声明队列等来说，Mnesia 也会这样做。虽然 Mnesia 在保持一致性方面工作得很好，但是当你需要执行某种类似路由键匹配的查询时会减慢速度。不过，其中的部分过程已经针对 direct 和 fanout 类型的交换器进行了优化，所以你不用承受 Mnesia 为了协调工作而付出的代价了。对于该交换器类型来说，绑定存储在 `ordered_set` 类型的 `rabbit_route` 表中。根据 ETS 表的介绍文档，针对这种类型表格的访问时间与数据库中的条目总数成对数关系。同时，由于 `ordered_set` 表格类型的特性，RabbitMQ 开发人员在从表格中选取数据时，可以执行一些有趣的优化来绕过 Mnesia。这意味着 RabbitMQ 路由表由 Mnesia 来提供对一致性的保证，而由普通 ETS 表来提供数据查询速度的保证。

direct 和 fanout 交换器

direct 交换器和 fanout 交换器之间的差别在于后者在查询 `rabbit_route` 表时忽略了路由键。因此，虽然你可以在队列绑定时为 fanout 交换器提供一个路由键，但是在路由消息的时候，该路由键会被忽略。当发布带有路由键的消息给 fanout 交换器时，该路由键同样会被忽略。

topic 交换器

topic 交换器则完全不同。这是因为存储的路由信息更加复杂。由于路由键可以包含以点（.）分隔的多个词，所以匹配消息路由键不仅仅是简单字符串的匹配。因此，RabbitMQ 实现了 `trie` 数据结构用来存储绑定路由键模式，以支持快速查询。以下两篇博客详细解释了 topic 交换器及其性能特点：http://www.rabbitmq.com/

1 更多有关ETS的信息请参阅http://erlang.org/doc/man/ets.html。

blog/2010/09/14/very-fast-and-scalable-topic-routing-part-1/ 和 http://www.rabbitmq.com/blog/2011/03/28/very-fast-and-scalable-topic-routing-part-2/。在第二篇文章中，你可以看到 RabbitMQ 的 topic 交换器实现，可以在 2.3GHz 的机器上，以 11 秒左右的时间，针对 2000 个模式匹配 1 000 000 个 topic。这说明 topic 交换器对于日常消息通信场景来说绰绰有余。记住，通常情况下，topic 交换器上的绑定相比 direct 或者 fanout 交换器来说，会占用更多的内存。

找到消息的目的地是一回事，投递消息则是另一回事儿。让我们看看在交换器处理消息路由信息之后发生了什么。

11.1.4 投递消息

在交换器找到消息需要路由的目的地之后，它会将目的地列表返回给 `rabbit_router`，并在之后着手将消息的副本投递到每一个目的地（队列或者交换器）。如果发布的消息中 `mandatory` 和 `immediate` 标记设置为 false 的话，那么这个过程会以异步的方式执行；并且从客户端的角度来看，服务器会变得更快。

事情开始变得有点棘手了，所以我们将该过程用图 11.1 展现出来。如果消息投递的目的队列是空的并且消费者准备好接收消息的话，那么该消息会直接发送给消费者，而不会经过队列这一步。你应该猜到了，这会极大地提升消息投递速度。下一个问题是消费者是否设置了 `auto-ack` 模式。如果消费者在订阅的时候将 `no-ack` 设置为 true 的话，那么服务器会忽略该消息；否则，该消息会被添加到 `pending-acknowledgment` 列表中用来记录消息。下一个问题是该消息是否路由到了持久化队列上，以及该消息是否以持久化的方式发布。如果确实如此，那么消息会写入磁盘，但被标记为已投递（already delivered）。因此当消息暂存队列的时候，它不会被发送给其他消费者了。

现在让我们回到 RabbitMQ 检查队列是否为空的那一刻。如果队列不为空的话，那么该消息就会入队。如果该消息不是持久化的话，那么它会被保存在内存中，当然前提是有足够的内存来存放消息。如果内存不足的话，那么消息会被写入磁盘，存放到瞬态存储中。如果是持久化消息的话，它们会被写入磁盘，并同时保存在内存中以提升消息投递速度。如果发生了内存不足的情况，那么消息会刷出到磁盘。通过这种方式，服务器既确保了消息的属性（例如持久化的作用），同时仍然尽可能快地投递消息。

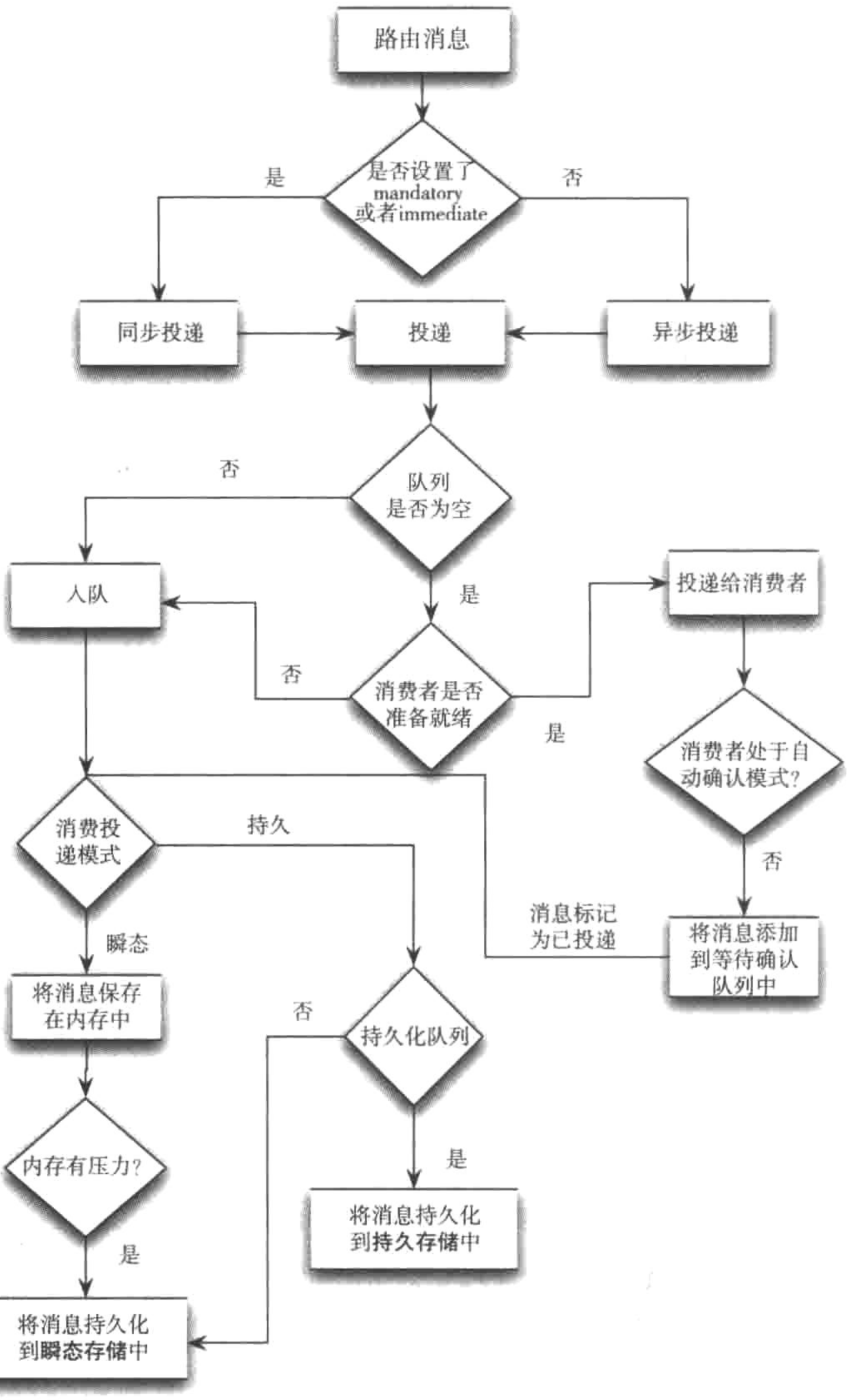

图 11.1 消息投递流程

如图 11.1 所示，当消息路由到了持久化队列的时候，如果它需要写入磁盘的话，就会进入持久存储；否则就会进入瞬态存储。如果 RabbitMQ 需要重启并恢复持久化队列的话，它只需看看持久存储中的内容，同时清除瞬态存储。

请记住，RabbitMQ 被优化为尽可能快地向消费者投递消息。如果你制定容量规划并计算消息的进出速率的话，那么你应该尽可能地让队列保持为空；虽然这不是最新发现，但它会让你拥有一个快节奏的服务器。但是如果消费者开始滞后，同

时队列开始填满的话，之后在某一时间点上，服务器就会收到内存告警，并开始将消息刷出到磁盘，而不管当时发布消息时使用的是什么参数。教训是时时刻刻关注队列大小。

在本节中，你看到了不同的算法和消息发布订阅设置如何影响整个系统的速度，以及像 auto-ack 模式标记设定能在很大程度上影响系统性能。在下一节中，我们将会看到硬件（内存）和 Erlang 虚拟机施加在 RabbitMQ 服务器上的硬性限制。

11.2 内存使用率和进程限制

当设计应用程序的时候，你通常会有两个基本限制：你选择的技术允许你做什么，以及当前硬件设定允许你做什么。在上一节中——贯穿本书——我们已经讨论过第一点了：不同的消息路由和分发算法如何影响设计决策。在本节中，我们会回顾硬件和 Erlang 虚拟机施加给 RabbitMQ 的硬性限制，以便你可以进行预先规划，同时看一下在单台机器上你可以将 RabbitMQ 的性能提升多少。举例来说，其中一个有趣的度量值是 RabbitMQ 创建的每个 AMQP 组件，例如队列、交换器和绑定需要多少内存。另一个需要考量的值是 RabbitMQ 为这些元素创建的 Erlang 进程数，因为 Erlang VM 对可以创建的进程总数有硬性限制。让我们来详细研究每一个元素，这样你在做容量规划计算的时候就有坚实的基础了。

11.2.1 内存使用率

首先要弄明白的是当声明队列的时候到底发生了什么。当声明队列时，RabbitMQ 会在不同的 Mnesia 表里添加多条记录，这取决于队列的类型。如果队列的声明中 durable 为 true 的话，那么就会在表 rabbit_queue 和表 rabbit_durable_queue 添加队列的记录。否则，只会在表 rabbit_queue 中存放记录。以上任何一张表的一条记录会占用大概 29 个字的大小。那意味着什么呢？在 Erlang 中字的大小依赖于你的操作系统。对于 32 位计算机来说字长为 4 字节；而在 64 位架构中，一个字占 8 个字节。我们说大约 29 个字的原因在于记录的大小同时还依赖于队列声明时采用的名称。以上过程如图 11.2 所示。

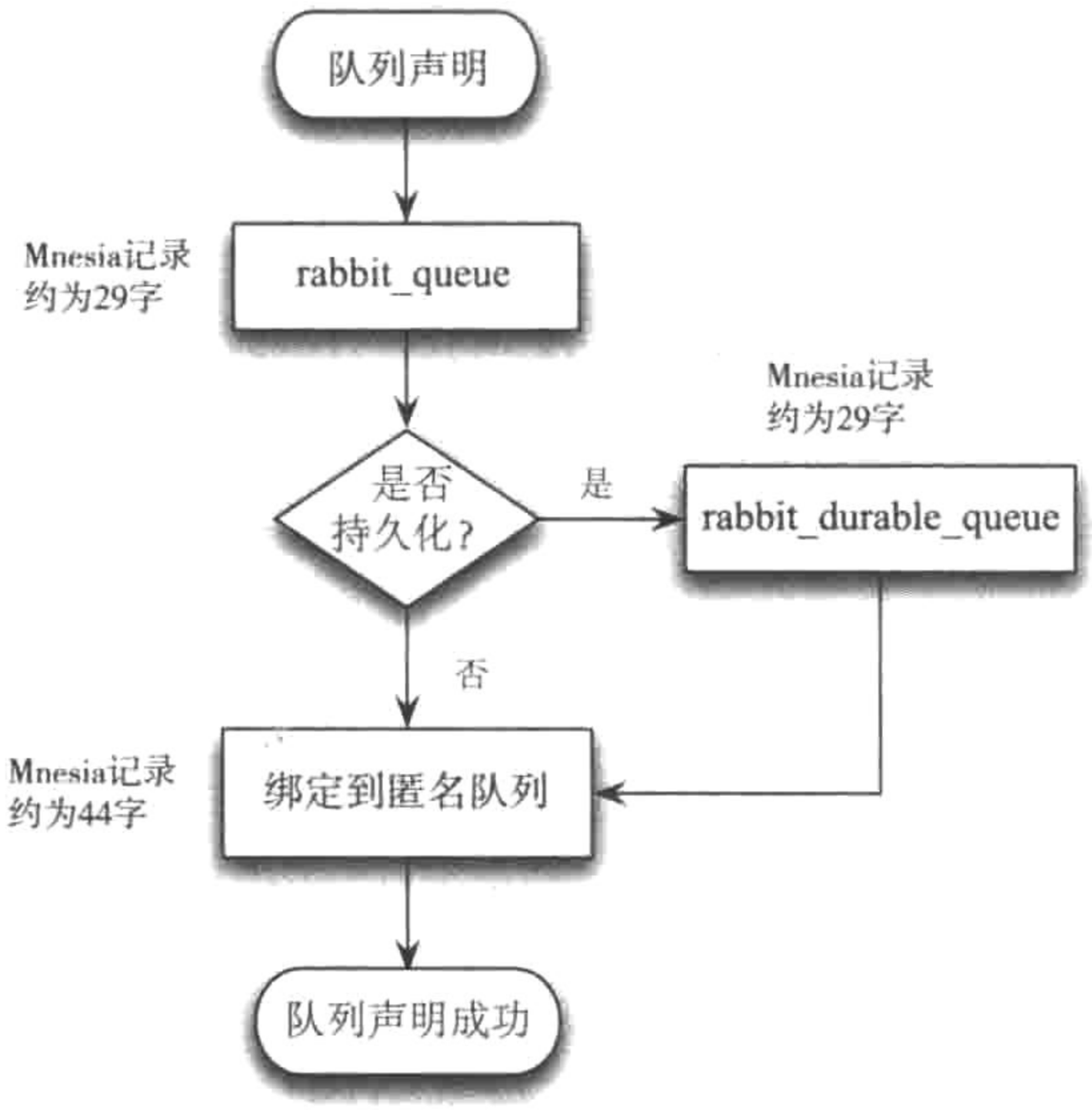

图 11.2 队列声明过程

根据 AMQP 规范，每个队列都会绑定到匿名交换器上。这意味着在队列声明之后，rabbit_route 表中也会有一条记录来保存队列和交换器之间的绑定。在其他 Mnesia 表中也可能会添加记录，这点稍后我们在讨论绑定规则的时候再说。rabbit_route 表中的一条记录会占用大概 44 个字的大小。

这意味着当你声明队列时，你会在 rabbit_queue 表中有一条 29 个字大小的记录，加上在 rabbit_route 表中 44 个字大小的记录。如果队列是持久化的（durable）话，那么也会在 rabbit_durable_queue 表中有一条 29 个字大小的记录。此外，在路由表中也会产生更多记录，你马上就会看到。表 11.1 详细地进行了解释。

表 11.1 队列元数据内存使用

	rabbit_queue	rabbit_durable_queue
持久化队列	x	x
瞬时队列	x	
字 / 记录	29	29

x 用于表示哪些表会为队列声明添加记录；同时，你可以看到这些表格中记录的字长。

当你声明交换器时，差不多也是这样，但却会更简单一些。当声明了新的交换器时，RabbitMQ 会在 rabbit_exchange 表创建一条大约占用 29 个字大小的记录。如果交换器是持久化的话，rabbit_durable_exchange 上也会创建一条记录。请参见表 11.2。

表 11.2 交换器元数据内存使用

	rabbit_exchange	rabbit_durable_exchange
持久化交换器	x	x
瞬时交换器	x	
字 / 记录	29	29

现在该来看看当你将队列绑定到交换器上的时候会发生什么。有两种情况：一种是将队列绑定到 direct 或者 fanout 交换器上，另一种是将队列绑定到 topic 交换器上。后者解释起来相对复杂，因此我们会在最后分析它。

当队列绑定到 direct 或者 fanout 交换器上的时候，RabbitMQ 会在至少两张 Mnesia 表上创建条目来记录绑定信息。这两张表分别是 rabbit_route 和 rabbit_reverse_route。记录大小为 44 个字。有几种组合会导致 RabbitMQ 在其他表上创建条目。这些组合取决于队列的持久化属性以及用于绑定的交换器。一个持久化队列绑定到一个瞬时交换器会导致在 rabbit_semi_durable_route 表上创建条目。最后，当你将持久化队列绑定到持久化交换器上的话，RabbitMQ 会在 rabbit_durable_route 表创建条目。这些条目每个大小均为 44 个字。在表 11.3 中你可以看到更详细的内容。

表 11.3 绑定元数据内存使用情况

	rabbit_route	rabbit_durable_route	rabbit_semi_durable_route	rabbit_reverse_route
持久化队列绑定到持久化交换器	x	x	x	x
持久化队列绑定到瞬时交换器	x		x	x
瞬时队列绑定到瞬时交换器	x			x
瞬时队列绑定到持久化交换器	x			x
字 / 记录	44	44	44	44

当绑定到 topic 交换器的时候，除了需要之前绑定到 direct 和 fanout 交换器时的所有元素外，还有额外的两张 Mnesia 表：rabbit_topic_trie_binding 和 rabbit_topic_trie_edge。顾名思义，绑定会存储在 rabbit_topic_trie_binding 表中。表中的一条记录大概占用 45 个字。topic 模式中有多少部分就会决定有多少条记录插入到 rabbit_topic_trie_binding 表中。因此像 a.b.c.d 这样的模式会在表中创建四条记录，每条占用 38 个字。

有了这些数据之后，你可以为消息通信应用做容量规划，并能够确定 RabbitMQ 使用的内存上限。如你所见，队列、交换器和绑定的内存占用很小。举个例子，在 64 位操作系统上，一个持久化队列绑定到一个持久化交换器会在 rabbit_queue 和 rabbit_durable_queue 表中占用 58 个字，另外会在各个 *_route 表中总共占用 176 个字。一共占用 234 个字，或者用字节表示是 234 字 ×8 字节 / 字，等于 1872 字节。

另一个施加在 RabbitMQ 上的硬性限制是每个 Erlang 节点的最大 Erlang 进程数。让我们来仔细看看吧。

11.2.2 Erlang 进程计数

可以在节点启动时指定 Erlang 节点上能够运行的最大 Erlang 进程数。这里指的是通过调用 ./sbin/rabbitmq-server -detached 来启动 RabbitMQ。RabbitMQ 的默认设置是每个 Erlang 节点 1048576，即 2^{20}，这应该绰绰有余了。

Erlang 应用程序在整个生命周期中会多次创建并销毁进程。例如，当 RabbitMQ 接收到 AMQP 客户端的 TCP 连接时，就会创建一个 Erlang 进程来管理该连接。同时，会有许多 Erlang 进程来处理 RabbitMQ 消息存储的逻辑。另一些进程会监控子进程来确保它们正常运作，等等。如果启动 RabbitMQ 服务器的话，大概会有 126 个进程在工作，这对于像 RabbitMQ 这样的服务器来说是非常少的。但是，如果你超过了默认的 2^{20} 限制会怎样？不幸的是，Erlang 会崩溃，RabbitMQ 也会崩溃。因此，你必须确保自己设置了合适的进程数。目前，2^{20} 个进程对大多数用户来说足够多了，并且也够用了，但是你的情况会不同。让我们看看什么情景下会使得进程数增加，好让你知道需要在容量规划计算中添加些什么。

RabbitMQ 的用户会通过以下事件来增加进程数：到服务器的新连接、创建新

的信道以及队列声明。一条新的连接会创建四个新的进程；同样地，在连接上开启一个新的信道会创建四个新的进程。队列的开销倒是最小的：只是每个队列一个进程。表 11.4 清楚地进行了展示。

表 11.4 连接、信道和队列使用的 Erlang 进程数

	进程数
新建连接	4
新建信道	4
队列声明	1

我们之前提过，大多数情况 2^{20} 个进程数绰绰有余了；不过对消息通信架构做一些数据规划总不会错。既然你已经充分了解了不同 AMQP 操作给 RabbitMQ 带来的影响，那么现在该来看看如何通过设置启用 SSL 来保障 RabbitMQ 的安全了。

11.3 SSL连接

当在公司封闭的网络环境中部署应用程序时，你几乎可以 100% 相信相关的各个参与方。如果 app_a 请求连接到你的 RabbitMQ 服务器的话，你可以简单地假定是由真正的 app_a 发起的请求。大部分情况下，我们没理由去怀疑这类请求；但是当你处理一些敏感数据，譬如信用卡信息时，你可能需要限制对应用程序特定区域的访问。由于不想泄漏这些信息，因此你需要用一种安全的方式来建立到 RabbitMQ 的加密连接，并传输数据。你可以使用 SSL[2] 协议在消息通信终端（譬如在消费者和生产者之间）传输数据。RabbitMQ 支持 SSL 开箱可用。因此对你来说，你需要设置所有的 SSL 组件来使用该基础设施。

在本节中，我们会解释如何通过 SSL 建立到 RabbitMQ 的安全连接。我们会使用 OpenSSL 库，它支持大多数 *nix 类操作系统，同时也支持 Windows。你可以在站点 www.openssl.org/ 上获取更多关于 OpenSSL 的信息。由于 OpenSSL 和安全是相当广泛的议题，因此我们无法在本章中详细讨论。如果你想要详细了解如何使用 OpenSSL 来保障你的网络安全的话，我们建议读一下这本书：*Network Security with OpenSSL*（Viega et al., 2002, O'Reilly Media）。

2 SSL代表的意思是安全套接字协议层。该加密协议允许安全的网络通信。

在下一节中，你会看到如何创建一个 SSL 认证机构并为你的客户端和服务器创建证书。这也就是我们经常提到的公钥基础设施（Public Key Infrastructure，PKI）。最后，你将使用这些证书来建立到 RabbitMQ 的 SSL 连接。

11.3.1 SSL 证书

以安全的方式来交换信息的其中一种手段是使用公钥密钥加密（public key cryptography）[3]。使用该技术，交换信息的双方拥有数学上相关的私钥和公钥，它们可以用于交换信息的加密与解密。公钥可以被公开出去，而私钥则必须保密。这种技术使用非对称算法，所以用来加密的密钥无法进行解密。如果用户 Bob 想要和 Alice 交换数据，那么他们交换各自的公钥。当 Alice 发送数据给 Bob 的时候，她会使用自己的密钥来加密该信息，然后 Bob 会使用 Alice 的公钥来解密数据。另一种公钥的用途是签名消息。消息的数字签名是根据私钥计算出来的；通过这种方式，接收方就可以用发送方的公钥来检测签名，从而可以判断该消息是否来自于对方。图 11.3 描述了该过程。

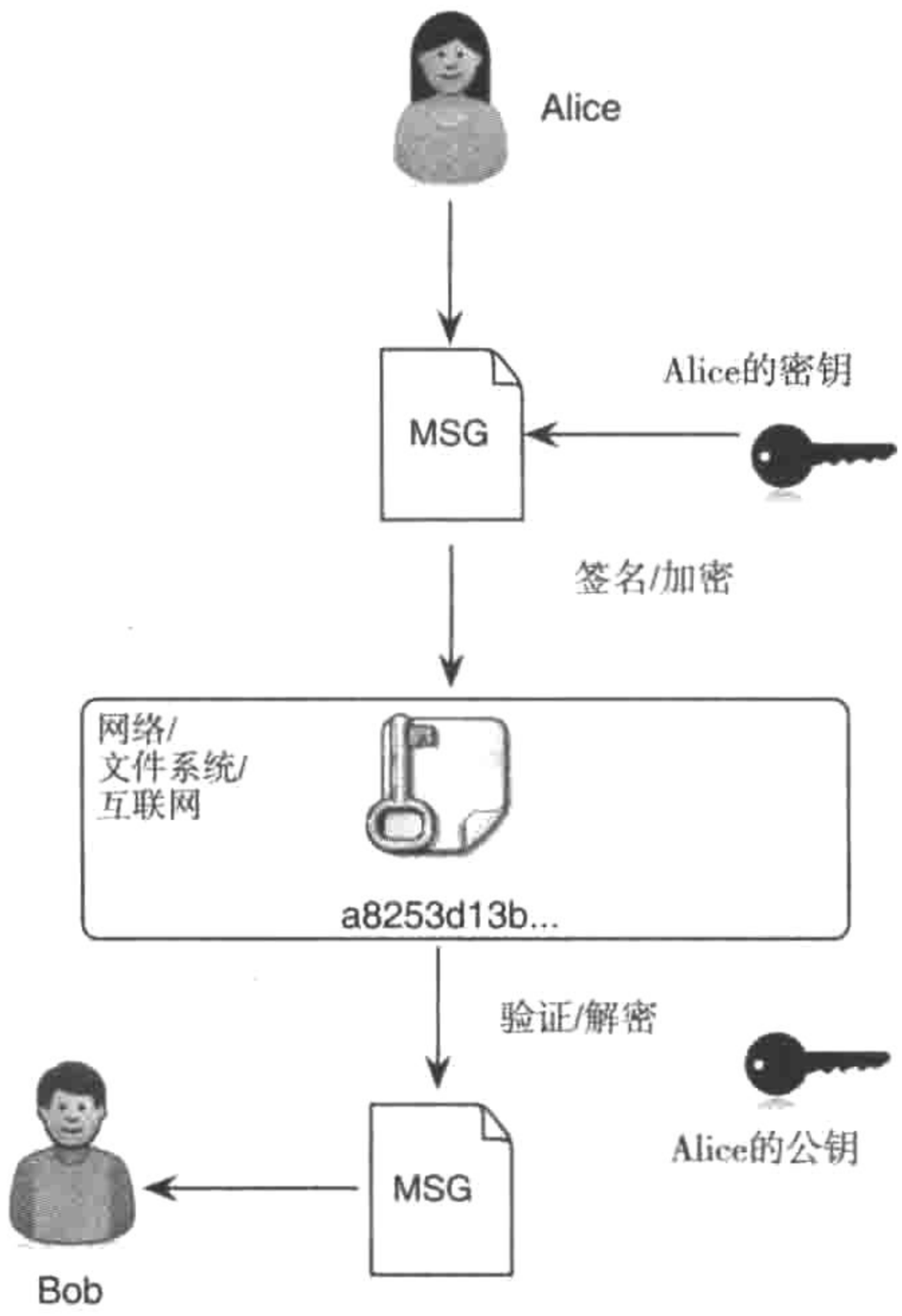

图 11.3 公钥加密

3 有关该主题的更多信息请参阅Wikipedia 上的条目：http://en.wikipedia.org/wiki/Public-key_cryptography。

获得对方的公钥是一回事儿，如何知道该公钥确实属于对方则是另一回事儿。为了确保密钥确实属于它的持有者，密钥会随同证明其真实性的证书（certificate）一同进行交换。证书是由可信的第三方作为认证中心（Certificate Authorities，CA）发布的。CA 负责确认密钥属于其所有者。虽然这听起来过于复杂，不过在 *Network Security with OpenSSL* 一书中有一个精彩的比喻，它将证书比作护照。护照上不仅有照片，而且还包含了一些个人信息。这些信息使得他人可以证明照片确属其人。当然你可以在地下室伪造一本护照，来尝试欺骗权威机构。为了防止这一点，你的护照上包含了政府发证机关的信息。对于你的护照来说，发证机关就是 CA 了。当你出国旅行时，入境办事处能够基于护照来辨认你，并能基于你所属政府添加的水印和其他方式来证明其真实性。

为了获取有效的 SSL 证书，你可以花钱请信任的第三方公司来颁发一个给你。如果你计划和公众交换数据的话，你可能会这样做。但是如果你只想要在组织内部共享数据的话，那么你可以设置自己私有的并且绝对信任的 CA。你将使用该 CA 来颁发证书给 RabbitMQ 和其消息通信客户端，用来交换数据。在下一节中，你将学习如何设置自己的 CA 并颁布证书给你的客户端和服务器，使它们之间能够建立 SSL 连接。

11.3.2 设置证书颁发机构

为了设置 CA，你将使用 `openssl` 命令行工具。该工具接收一大堆很难记忆的选项。为了方便起见，`opessl` 可以使用配置文件，而你可以在其中使用键值对来指明选项。你也可以通过检查该配置文件来追踪曾经用于生成证书的参数。让我们从为你的 CA 创建基本环境开始吧。

> **注意**：如你所知，RabbitMQ 运行在 Erlang 之上，所以尝试在应用程序中使用 OpenSSL 之前，你需要先让 Erlang 支持它才行。如果你在搭建 Erlang 之前就安装好了 OpenSSL 的话，那么你可能马上就能使用它了。如果你没有在系统中安装 OpenSSL 的话，那么你需要先安装它，然后再重新安装 Erlang，同时启用 OpenSSL。

你需要创建文件夹来存放证书和 CA 配置文件。开启终端窗口，然后输入以下命令：

```
$ mkdir rmqca
$ cd rmqca
$ mkdir certs private
$ chmod 700 private
```

首先你创建了 rmqca 文件夹来存放文件。我们使用名称 rmqca 来表示 RabbitMQ 认证中心。你可以随意更改该名称来适应你所在组织的需求。之后，你还将创建两个文件夹：certs 和 private；前者用于存放 CA 产生的证书，后者用于存放 CA 密钥。记住，一定不能向第三方泄漏 CA 私钥；因此使用 chmod 将 private 文件夹的访问权限设置成只允许当前用户访问。

为了能生成证书，OpenSSl 还需要一些文件。由于 CA 创建的证书无法共享序列号，因此你需要创建一个文件来存放最近证书的序列号。每颁发一份新的证书，OpenSSL 会负责递增该数字。OpenSSL 期望该数字为十六进制，并且包含至少两位。所以当你首次创建该文件的时候，你需要为数字在左侧添加一个 0，就像这样：

```
$ echo 01 > serial
```

还有最后一个文件需要创建，OpenSSL 会用该文件存放 CA 颁发的证书。我们会将该文件命名为 index.txt；由于尚未创建任何证书，因此该文件是空的。你可以通过下列命令创建它：

```
$ touch index.txt
```

在拥有了 OpenSSL 工作的基本环境之后，你需要创建配置文件。该文件用于为各种 openssl 命令提供默认值。你会用它来创建自己的 CA 并在稍后用来颁发证书。你将会看到，该配置文件为每个命令分成了若干部分，因而也更便于理解。其中一个命令叫作 ca，它用于设置 CA 和证书废弃列表（Certificate Revocation List，CRL）[4]。让我们在 rmqca 文件夹下创建一个叫作 openssl.conf 的文件，然后把下面的 CA 命令内容添加进去。

4 证书废弃列表用于在CA发布的证书过期时通知客户。客户端可以从CA下载该列表，然后拒绝CRL中过期的证书。

清单 11.1 openssl.conf CA 命令配置

```
[ ca ]
default_ca = rmqca

[ rmqca ]
dir = .
certificate = $dir/cacert.pem
database = $dir/index.txt
new_certs_dir = $dir/certs
private_key = $dir/private/cakey.pem
serial = $dir/serial

default_crl_days = 7
default_days = 365
default_md = sha1

policy = rmqca_policy
x509_extensions = certificate_extensions

[ rmqca_policy ]
commonName = supplied
stateOrProvinceName = optional
countryName = optional
emailAddress = optional
organizationName = optional
organizationalUnitName = optional

[ certificate_extensions ]
basicConstraints = CA:false
```

❶ 该部分的名称

❷ CA配置

❸ 过期配置

❹ 政策配置

❺ 扩展部分

在该文件中，你为 ca 命令提供了默认选项，你仍然可以在调用 openssl 命令时，通过命令行进行覆盖。可以看到，.conf 文件中的每一部分是用放在一对方括号（[]）内的头进行标记的。配置文件是由一系列配置选项的键值对构成的。从声明 ca 部分❶开始，将默认 CA 的名称设置为 rmqca。然后 OpenSSL 会查找该名称所对应的部分，之后加载配置选项的剩余部分。

在 rmqca 部分❷中，你设置了一个叫作 dir 的变量，指向和 .conf 文件同样的目录。接下来的几行都会引用该变量，因而你不需要每次都输入完整的路径。你告诉 openssl 证书会存放在和配置文件相同目录下的 cacert.pem 文件中。然后你将数据库设置为 index.txt。CA 颁发的证书会存放在 certs 文件夹下；你的私钥需要从 private/cakey.pem 文件中提取，同时用于生成证书的序列号会存储在你之前创建的 serial 文件中。

然后你需要为你的证书配置过期时间❸。此处证书会在一年后过期，你会每隔

七天提供一个 CRL 文件，并且使用 sha1[5] 作为哈希函数来生成证书。

在文件的下一部分中你配置了 CA 策略❹，告诉 openssl 在证书中的哪些字段是必填的。对该 CA 来说，commonName 是必需的，而其他字段像 countryName 或 emailAddress 是可选的（更多详细信息，请参阅 http://www.openssl.org/docs/apps/ca.html#POLICY_FORMAT）。在该部分中，你指明了 CA 支持哪些 x509[6] 扩展，这些扩展在配置文件的 certificate_extensions 部分中❺。此处，你将 basicConstraints 设置为 CA:false。从根本上来说，这意味着 CA 颁布的证书无法将自己作为 CA——也就是说它们无法用于签名和颁发新证书。

现在你需要配置 req 命令来生成证书。往 openssl.conf 添加一些新的键值对。让我们把下列内容添加到 openssl.conf 文件中。

清单 11.2 openssl.conf req 命令配置

```
[ req ]                                                    ← ❶ req配置部分
default_bits = 2048
default_keyfile = ./private/cakey.pem
default_md = sha1
prompt = yes
distinguished_name = root_ca_distinguished_name
x509_extensions = root_ca_extensions

[ root_ca_distinguished_name ]
commonName = hostname

[ root_ca_extensions ]                                     ← ❷ 根扩展
basicConstraints = CA:true
keyUsage = keyCertSign, cRLSign

[ client_ca_extensions ]                                   ← ❸ 客户端扩展
basicConstraints = CA:false
keyUsage = digitalSignature
extendedKeyUsage = 1.3.6.1.5.5.7.3.2

[ server_ca_extensions ]                                   ← ❹ 服务器端扩展
basicConstraints = CA:false
keyUsage = keyEncipherment
extendedKeyUsage = 1.3.6.1.5.5.7.3.1
```

在该部分❶中，通过指明你想要为证书生成 2048 位的密钥来配置 req 命令。如果想要让你的密钥安全的话，这是建议的最小数字；该密钥会写入 ./private/

5 更多有关SHA1的信息请参阅http://en.wikipedia.org/wiki/SHA-1。

6 可以在OpenSSL网站上找到更多有关x509扩展的信息： http://www.openssl.org/docs/apps/x509.html。

cakey.pem 文件，并使用 sha1 作为默认的哈希函数。通过将 prompt 设置为 yes，你告诉 req 命令，每当它需要填写 distinguished_name 下的值的时候，都应该提示你。你可以看到，distinguished_name 的内容在另一个叫作 root_ca_distinguished_name 的部分中，在那里你将 hostname 设置为了 commonName。

在接下来的这一部分中，你为根证书提供了一些扩展❷。在本例中，你将根证书设置为可以用于签名其他证书（这是整个设置中最为关键的一点）。当设置客户端扩展时❸，客户端证书无法像 CA 那样使用，但是可以用于往来数据的签名。extendedKeyUsage 字段有一串特别的数字[7]，用于说明该证书可以用于客户端认证。最后是服务器端扩展部分❹，在该示例中你想让证书用于加密数据及认证服务器。如你所见，你用不同的一串值来设置 extendedKeyUsage 字段。

通过这些设定，你完成了 OpenSSL 的环境设置，并准备好创建证书了。虽然该过程比较复杂，但你只需做一次即可。将配置选项放入该文件的好处就是，你不需要用心记住所有这些设置。让我们继续前进到下一节来看看如何创建自己的证书。

11.3.3 生成根证书

首先需要做的是生成一份 CA 证书。这是一份你绝对信任的证书。所有其他的证书都将派生于该证书，并且它将用来在不同应用之间建立信任链（chain of trust）。让我们运行下列命令：

```
$ openssl req -x509 -config openssl.conf -newkey rsa:2048 -days 365 \
  -out cacert.pem -outform PEM -subj /CN=RMQCA/ -nodes
```

你应该看到像这样的输出：

```
Generating a 2048 bit RSA private key
.....................................................+++
.....................................................
.....................+++
writing new private key to './private/cakey.pem'
-----
```

使用该命令创建的证书会存储在文件 cacert.pem 中。格式为 PRM，使用 x509 扩展。该密钥会使用 rsa:2048 进行加密，并在 365 天内有效。有关 req 命

7 如果想知道更多细节，我们再一次推荐这本由Viega等人编写的*Network Security with OpenSSL*。OpenSSL是一个非常广阔的议题。如果你想深入研究的话，那么最好先把这本书吃透。

令和其选项的明确信息，请参考 http://www.openssl.org/docs/apps/req.html。

下一步要做的是创建同样的证书，但是使用的是 DER 格式，微软产品更倾向于使用该格式。运行下列命令：

```
$ openssl x509 -in cacert.pem -out cacert.cer -outform DER
```

现在你有了自己的根证书了，该创建客户端和服务器端的证书了。

11.3.4 生成服务器端证书

为了生成服务器端证书，你需要创建一个文件夹来存放它。之后生成服务器密钥，最后使用根证书来认证它。

```
$ cd ..
$ mkdir server
$ cd server
$ openssl genrsa -out key.pem 2048
Generating RSA private key, 2048 bit long modulus
.....................................+++
.....................................
.........................+++
e is 65537 (0x10001)
```

首先移动到包含 rmqca 的文件夹下，创建 server 文件夹，然后进入该文件夹。调用 openssl 命令来生成 RSA 密钥。下一步是为该密钥创建一个证书请求：

```
$ openssl req -new -key key.pem -out req.pem -outform PEM \
  -subj /CN=$(hostname)/O=server/ -nodes
```

现在该证书请求可以被 CA 用来为你的 RabbitMQ 服务器提供证书。让我们创建服务器证书：

```
$ cd ../rmqca/
$ openssl ca -config openssl.conf -in ../server/req.pem -out \
  ../server/cert.pem -notext -batch -extensions server_ca_extensions
Using configuration from openssl.conf
Check that the request matches the signature
Signature ok
The Subject's Distinguished Name is as follows
commonName            :ASN.1 12:'mrhyde'
organizationName      :ASN.1 12:'server'
Certificate is to be certified until Oct 5 23:10:35 2012 GMT (365 days)

Write out database with 1 new entries
```

```
Data Base Updated
```

你首先进入 rmqca 文件夹下，并通过使用 openssl.conf 文件运行 openssl ca 命令。输入的文件是之前创建的 req.pem，同时输出的文件是 cert.pem 证书，将会存放在你的 server 文件夹内。现在你需要重复同样的过程来创建客户端证书。

11.3.5 生成客户端证书

考虑到整个过程非常相似，这次我们不会很详细地进行解释。就像之前的示例一样，首先为客户端创建一个密钥，然后根据该密钥生成一个证书请求，最后将证书请求提交给 CA 来颁发客户端证书，并存放在 client 文件夹下的 cert.pem 中。以下展示了需要运行的命令，以及这些命令产生的输出，你的计算机上看到的可能会有所不同：

```
$ cd ..
$ mkdir client
$ cd client
$ openssl genrsa -out key.pem 2048
Generating RSA private key, 2048 bit long modulus
.....................................................
.....................................................+++
.........................+++
e is 65537 (0x10001)

$ openssl req -new -key key.pem -out req.pem -outform PEM \
  -subj /CN=$(hostname)/O=client/ -nodes

$ cd ../rmqca/
$ openssl ca -config openssl.conf -in ../client/req.pem -out \
  ../client/cert.pem -notext -batch -extensions client_ca_extensions
Using configuration from openssl.conf
Check that the request matches the signature
Signature ok
The Subject's Distinguished Name is as follows
commonName            :ASN.1 12:'mrhyde'
organizationName      :ASN.1 12:'client'
Certificate is to be certified until Oct 5 23:14:50 2012 GMT (365 days)

Write out database with 1 new entries
Data Base Updated
```

现在你解决了证书的问题，该对 RabbitMQ 进行配置，以便在接收连接时使用 SSL。值得注意的是如果你查看 rmqca 文件夹下的 serial 和 index.txt 文件的内容的话，就会看到 serial 文件现在是数字 03 了，因为到目前为止你已经生成

了三份证书了。同时，index.txt 文件列出了你颁发过的证书。

11.3.6 启用 RabbitMQ 的 SSL 监听器

为了启用 RabbitMQ 的 SSL 功能，你需要在 rabbitmq.config 文件内添加一些配置内容。如果你还没有创建过该文件，那么现在是时候来创建了。该文件的位置因操作系统和 RabbitMQ 的发行版本的不同而不同。举个例子，在通用的 UNIX 环境中，该文件位于 /etc/rabbitmq 文件夹内。想要知道更多关于当前设置下该文件的位置信息的话，你可以参考 RabbitMQ 在线文档 http://www.rabbitmq.com/configure.html#config-location。

在创建该文件之后，你需要往里面添加两个条目：ssl_listeners 和 ssl_options。前者会为传入的连接启用 TCP 监听器；而后者会告诉 RabbitMQ 在哪里可以找到服务器证书，同时需要对通过 SSL 连入的客户端施加怎样的认证需求。如果在系统上已经有了该文件的话，就在文件末尾进行添加即可。配置文件应该看起来像下列清单所示。

清单 11.3 启用 RabbitMQ 的 SSL 功能

```
[
  {rabbit, [                                                    <-- ❶ 启用SSL监听器
     {ssl_listeners, [5671]},
                                                                <-- 配置SSL
     {ssl_options, [{cacertfile,"/path/to/rmqca/cacert.pem"},
                    {certfile,"/path/to/server/cert.pem"},
                    {keyfile,"/path/to/server/key.pem"},        <-- ❷ 客户端验证
                    {verify,verify_peer},
                    {fail_if_no_peer_cert,false}]}
   ]}
].
```

如你所知，rabbitmq.config 文件使用 Erlang 语法来指明配置选项。首先你设置了 ssl_listeners ❶来说明 RabbitMQ 会在端口 5671 上监听 SSL 连接。然后你设置了 ssl_options，RabbitMQ 会用该选项来对 Erlang 的 new_ssl 应用进行配置[8]。

这里的三个选项 cacertfile、certfile 和 keyfile，顾名思义：cacertfile

8 更多有关new_ssl应用程序的详情，请参阅在线文档http://www.erlang.org/documentation/doc-5.7.5/lib/ssl-3.10.8/doc/html/new_ssl.html。在那里可以找到所有选项的说明。

是你的 CA 私有证书，certfile 是服务器自己的证书，而 keyfile 就是服务器的密钥了。它们会被客户端用来认证服务器，同时也用来确认客户端的真实性。最后一个参数是 fail_if_no_peer_cert ❷，它告诉服务器如果客户端发送证书过来的话，服务器需要验证其真实性（verify_peer）；但是如果客户端没有发送证书的话，那么服务器不会拒绝客户端（fail_if_no_peer_cert， false）。记住当你使用 SSL 来建立加密连接时——因此，服务器会将证书展现给客户端——但是你也许会想要服务器能通过请求客户端的证书来建立信任链。这和浏览器的工作机制相似，服务器发送证书，但是浏览器却不需要将你的证书发送回去（当浏览 Web 的时候，大多数人都没有自己的 SSL 证书）。

现在是时候将该设置应用到 RabbitMQ 上了。重启 RabbitMQ（如果没有运行的话，就启动它），你应该就能看到 rabbit.log 中的下列条目，说明新的 SSL 监听器已经启动了：

```
$ tail -f /var/log/rabbitmq/rabbit@host.log
=INFO REPORT==== 9-Oct-2011::17:52:08 ===
started TCP Listener on 0.0.0.0:5672

=INFO REPORT==== 9-Oct-2011::17:52:08 ===
started SSL Listener on 0.0.0.0:5671
```

下一步要做的是通过使用 PHP 客户端库连接到 RabbitMQ 来测试这些新的配置。

11.3.7 测试你的 RabbitMQ SSL 设置

为了测试 SSL 的设置，你将使用 php-amqplib 库来连接 RabbitMQ。PHP SSL 实现需要客户端密钥、证书和 CA 证书这三者都在相同的文件中，因此你将为 PHP 来创建这样的文件。打开命令行并输入以下命令：

```
$ cat client/key.pem > phpcert.pem
$ cat client/cert.pem >> phpcert.pem
$ cat rmqca/cacert.pem >> phpcert.pem
```

你的 PHP 客户端将使用 phpcert.pem 文件来处理 RabbitMQ 的 SSL 连接。让我们编写客户端代码。添加下列内容到 ssl_connection.php 文件中。

清单 11.4 PHP OpenSSL 连接

```
<?php
require_once(__DIR__ . '/path/to/php-amqplib/amqp.inc');

define('HOST', 'localhost');                                    ← ❶ 连接选项
define('PORT', 5671);
define('USER', 'guest');
define('PASS', 'guest');
define('VHOST', '/');
define('AMQP_DEBUG', true);

define('CERTS_PATH',
  '/path/to/ca/folder/');

$ssl_options = array(                                           ← ❷ SSL选项
      'cafile' => CERTS_PATH . '/rmqca/cacert.pem',
      'local_cert' => CERTS_PATH . '/phpcert.pem',
      'verify_peer' => true
);

$conn = new AMQPSSLConnection(HOST, PORT, USER, PASS,           ← ❸ 创建连接
                              VHOST, $ssl_options);

function shutdown($conn){                                       ← ❹ 连接清除
    $conn->close();
}

register_shutdown_function('shutdown', $conn);
while(1){}
?>
```

如果你运行这段脚本的话，将会看见 AMQP 库的调试信息，会向你展现它是如何同 RabbitMQ 协商连接的。在成功建立连接之后，如果你对 RabbitMQ 日志文件运行 `tail` 命令的话，就应该看见与下列所示类似的消息，它向你展示了 RabbitMQ 与你的 PHP 脚本建立连接并且该连接升级成为 SSL 协议：

```
$ tail -f /var/log/rabbitmq/rabbit@host.log
=INFO REPORT==== 9-Oct-2011::21:01:03 ===
accepted TCP connection on 0.0.0.0:5671 from 127.0.0.1:64940

=INFO REPORT==== 9-Oct-2011::21:01:03 ===
starting TCP connection <0.16304.0> from 127.0.0.1:64940

=INFO REPORT==== 9-Oct-2011::21:01:03 ===
upgraded TCP connection <0.16304.0> to SSL
```

让我们看看这段脚本做了什么。首先像往常一样包含了 AMQP 库，然后声明了用于连接配置的常量❶。这里需要重点关注的是，你为 SSL 连接使用了端口 5671，

来替代使用默认的 RabbitMQ 端口（5672）。之后你为 PHP 创建了 SSL 选项数组❷，因此它就可以知道如何找到密钥和证书信息了。通过使用 AMQPSSLConnection 类开启连接❸，并随同 SSL 选项传入了一共六个参数。最后，你通过设置 shutdown 函数来准备对连接进行清理❹。该函数会在脚本结束后（例如当你按下 ctrl-c 组合键关闭脚本时）负责关闭连接。

至此，我们结束了 RabbitMQ SSL 的讨论。现在你可以在服务器和其消息通信客户端之间建立连接了，同时双方均能认证对方的真实性。当开启服务器端的连接时，通过检测服务器端证书，你能确保消息的来源是可靠的。同时，服务器可以验证客户端的真实性，因此它不会接收来自非受信方的连接。

我们已经说过很多遍了，这是一个很大的议题，超越了本书的范畴。需要注意的是，取决于你使用的语言，你将不得不去了解具体如何在你的平台上开启 SSL 连接。因为实现方面的细节会由于供应商的不同而不同。记住在本节中我们准备的 PKI 设置不仅仅可以用于 RabbitMQ，也可以用于其他许多地方。CA 并不仅限于认证 RabbitMQ 服务器和其客户端；你也能用它来认证其他应用程序之间的通信。举例来说，你可以为自己的内部网络颁发证书，来为公司内部网站启用安全网络浏览（使用 HTTPS）。

11.4 总结

在本章中，我们讨论了一些有趣的议题，譬如性能、容量规划，以及安全。当谈及提升 RabbitMQ 性能时，很清楚的是已经没有什么秘密可言了。所有这一切都依赖于你的使用场景；你必须始终权衡利弊。如果想要发挥 RabbitMQ 的性能，那么你就需要明智地分析不同交换器类型提供的路由算法的利与弊。此外，你已经见识到了像消息持久化、队列持久化和消费者确认模式这些属性的组合会影响到消息通过服务器的路径，这会影响到应用程序性能。考虑到容量规划，为了计算消息通信结构的内存使用，你必须考虑使用的队列或者交换器种类的因素，以及哪些 AMQP 元素会在服务器上启动 Erlang 进程。虽然 Erlang 拥有很高的进程限制上限，但它不是无限的，所以通过做一些数学计算，你可以得到匹配你需求的数值。最后，我们讲了像 OpenSSL 这样宽泛的议题的一些要点。你把服务器配置成可以接受 SSL

连接并对客户端进行认证。为了测试你的设置，你用 PHP 客户端通过 SSL 来连接服务器，但是让 PHP 接收你的证书还是花了不少力气。这样奇怪的事情发生在使用 OpenSSL 的几乎所有平台上。

在下一章中，我们会进入 RabbitMQ 插件的世界当中。你将看到自己可以获取哪些插件来增加服务器的能力。但不仅于此，你还要构造自己的插件。系好你的“安全带”，因为下一章你就要用 Erlang 编程了。你没听错：Erlang。

12 聪明的Rabbit：扩展RabbitMQ

本章要点

- 安装RabbitMQ插件
- 回顾那些有趣的插件
- 实现自定义交换器插件

到目前为止，你已经学习了如何将 RabbitMQ 作为 AMQP 消息服务器，使用其中附带的开箱即用功能。在第 8 章中，你也看到了需要定制一些工具来更方便地管理服务器。你启用了 Management 插件，它包含了一个灵巧的 Web 界面，实现了许多服务器的管理功能。启用插件的过程十分简单：只需在 shell 上运行命令：`rabbitmq-plugins enable rabbitmq_management`，该插件就可以正常使用了。想想看，如果你能将自定义行为添加到服务器上的话，岂不妙哉？

在本章中，我们会深入理解 RabbitMQ 插件，看看你如何处理它们，以及它们能带给我们哪些功能。你将学习如何启用插件；同时，如果不再需要它们提供的功能的话，你该如何卸载它们。你也许会想：从哪儿可以获取 RabbitMQ 的插件呢？别担心，我们会告诉你的。已经有许多非常棒的插件，你将看到如何获取它们。

当然，仅仅学习现存的插件还远远不够。在 12.2 节中，你将制作自己的插件。

是的，你没听错。你将动手使用 Erlang 来编程并创建你自己的插件。如果你不懂 Erlang 也别担心；我们会介绍足够的内容，以便让你开始编程。但是如果你了解 Erlang 的话，那写起来就舒心多了。那么就让我们进入下一节，给我们的 Rabbit 加把油吧！

12.1 RabbitMQ插件

如果你观察过像 RabbitMQ 这样的系统，就会发现随系统一同发布的功能对大多数用户来说都很有用。对于服务器默认配置或者是服务器新版本发布时的新特性来说，也是一样的。但是当你需要某些功能而服务器没有的时候，该怎么办呢？针对这种情况，RabbitMQ 可以通过添加插件的方式来进行增强。你可以从网络上找到插件，或者也可以自行编写。首先让我们看看什么情况下你会需要插件，然后我们来看看你该从哪里来获取服务器插件。

12.1.1 你可以用插件做什么

那么，哪些使用场景下，RabbitMQ 提供的开箱即用功能无法实现呢？以下是一些可能的场景或者需求列表，我们可以通过安装插件的方式来解决：

- 支持 AMQP 以外的协议
- 不同的认证机制（LDAP，自定义数据库）
- 消息复制
- 新的交换器和路由算法
- 消息日志和审计

让我们来看一些例子吧。

不同的协议——STOMP

人们进行着大量实验来为 RabbitMQ 添加更多协议的支持。众所周知，AMQP 是 RabbitMQ 支持的默认协议，但这并不能满足所有人的需要，于是就有了 STOMP 协议插件。STOMP 是一个简单的基于文本的协议，用于在应用之间传输数据。STOMP 的一个优势是，它能与其他服务器一起工作，如 ActiveMQ。如果你的代码基于 ActiveMQ 和 STOMP，并且你想只使用 RabbitMQ 的话，那么可以通过使用

STOMP 插件来开始一步步的迁移。或者你使用的编程语言没有 AMQP 客户端，但是却支持 STOMP 的话，那么你可以通过安装该插件来开始使用 RabbitMQ。如果你想要学习更多有关 STOMP 的知识的话，则可以访问：http://stomp.github.com/。

LDAP 认证

另一个 RabbitMQ 插件的使用场景是，针对不同于简单 AMQP 来进行对连接到服务器的认证需求。假设你的系统中所有的用户管理均通过 LDAP，并且你也想在连接到 RabbitMQ 时继续使用它。多亏了 LDPA 认证后端，使得这成为可能：http://www.rabbitmq.com/plugins.html#rabbitmq-auth-backend-ldap。安装该插件，添加合适的配置，你就能畅通无阻地使用 LDAP 认证了。

自定义交换器类型

新的协议和不同的认证机制这两者并不是你唯一可以添加到 RabbitMQ 之上的东西。你可以步入 AMQP 底层来实现自定义路由规则的交换器。一个有趣的自定义交换器例子是由 Jon Brisbin 实现的 Riak 交换器（https://github.com/jbrisbin/riakexchange）。Riak 是一个受 Dynamo 启发的键值存储，提供开箱即用的容错能力。如果为了审计的目的，你需要记录经过 RabbitMQ 交换器的每一条消息的话该怎么做呢？一种方法是使用 fanout 交换器，并绑定一个额外的队列，然后使用一个普通的 AMQP 消费者来实现日志；或者你可以使用 Riak 交换器来让服务器为你处理所有这些：将每一条消息都放入一个 Riak 桶（bucket）[1]。

如果你需要将消息从一台服务器复制到距离数据中心几英里之外的另一台服务器上的话，你该怎么做呢？针对这种情况，我们可以使用 RabbitMQ Shovel 插件。在该插件的配置文件里提供一个队列名称和远程服务器上的目的交换器，Shovel 会负责将消息传输到远程交换器上的。

能用插件做的不止这些。你不必将自己局限于添加新的交换器类型或者认证方式。你可以使用插件来做任何 Erlang 允许你做的事情；你需要的是发挥自己的想象力和判断能力。毕竟，你不会想让服务器因为你的异想天开的实现而崩溃。现在你知道了 RabbitMQ 插件可以用来做什么了，让我们看看在哪里可以找到它们吧。

---

1 Riak中的bucket是一种组织数据的方式，类似于SQL数据库中的表。

12.1.2 在哪里可以找到插件

第一处是网站上专门为 RabbitMQ 插件开辟的页面：www.rabbitmq.com/plugins.html。在这里你可以找到维护插件（maintained plugins）列表和另一个实验插件（experimental plugins）列表。前者是由 RabbitMQ 团队维护并且与新版服务器保持更新的。同时你可以通过 RabbitMQ 邮件列表在 http://lists.rabbitmq.com/cgi-bin/mailman/listinfo/rabbitmq-discuss 提交 bug 和需求。虽然针对实验插件你也可以这样做，不过你提的问题可能没人来回答。

12.1.3 安装插件

你已经看到自从 RabbitMQ 2.0.7 版以来，安装插件变得相当简单了：只需运行诸如 `./rabbitmq-plugins enable rabbitmq_management` 之类命令，重启服务器，这样就行了。插件就可以使用了！现在的问题是，如果你想启用的插件不是服务器发行中的一部分该怎么办呢？首先，你得下载插件的 `.ez` 文件到 RabbitMQ 安装目录的 `plugins` 文件夹下[2]，之后像往常一样运行 `./rabbitmq-plugins enable plugin_name` 命令即可。

让我们通过启用 STOMP 插件来试试这些指令吧。进入 RabbitMQ 安装目录下的 `sbin` 文件夹，并输入以下指令：

```
$ ./rabbitmq-plugins enable rabbitmq_stomp
The following plugins have been enabled:
  rabbitmq_stomp

Plugin configuration has changed. Restart RabbitMQ
for changes to take effect.
```

现在，如果你已经启动服务器的话，就将它停止：

```
$ ./rabbitmqctl stop
Stopping and halting node rabbit@mrhyde ...
...done.
```

然后重新启动服务器来加载新的插件：

```
$ ./rabbitmq-server -detached
```

---

2 如果没有`plugins`文件夹的话，那就在RabbitMQ安装目录下创建一个。之后如果你不再需要使用插件的话，也可以安全地删除。

```
Activating RabbitMQ plugins ...
7 plugins activated:
* amqp_client-2.7.0
* mochiweb-1.3-rmq2.7.0-git
* rabbitmq_management-2.7.0
* rabbitmq_management_agent-2.7.0
* rabbitmq_mochiweb-2.7.0
* rabbitmq_stomp-2.7.0
* webmachine-1.7.0-rmq2.7.0-hg
```

如你所见，在那些已经安装的插件之中，服务器将 rabbitmq_stomp-2.7.0 插件和其他已激活的插件一起列了出来。STOMP 插件会使用默认配置运行。为了测试它是否如期工作，让我们使用默认的用户凭证和 STOMP 协议，连接 RabbitMQ。让我们使用 nc（netcat 命令行工具）命令来完成之前提到的任务吧：

```
$ nc localhost 61613
```

之前的命令会开启到服务器的连接。现在你输入的所有内容都会被发往 STOMP 适配器，它会尝试解析帧。现在输入以下内容来启动会话：

```
CONNECT
login:guest
passcode:guest
^@
```

在输入凭证之后，你需要添加一行额外的空白行，代表一个空的内容，然后输入 ^@（按下 Control 键和 @ 符号），这代表了帧的末尾。如果一切正常的话，你应该会得到如下应答：

```
CONNECTED
session:session-ds/mGfvEV6TkPXPVcUv8YA==
heart-beat:0,0
version:1.0
```

通过这个简单的例子，我们结束了 STOMP 插件的测试。我们的目的是安装插件并在零配置下将其启动并运行。你可以通过输入 ^c（Ctrl+C 组合键）来退出会话。如果你想要学习更多有关 RabbitMQ 下的 STOMP 如何使用，则可以参考该文档：http://www.rabbitmq.com/stomp.html。

12.1.4 移除插件

现在，假设你不再需要 STOMP 插件了，想要移除它。通过一直使用的

rabbitmq-plugins 命令就可以轻松完成任务。首先在 sbin 文件夹下运行下列命令来禁用该插件：

```
$ ./rabbitmq-plugins disable rabbitmq_stomp
The following plugins have been disabled:
  rabbitmq_stomp

Plugin configuration has changed. Restart RabbitMQ
for changes to take effect.
```

然后需要停止服务器：

```
$ ./rabbitmqctl stop
Stopping and halting node rabbit@mrhyde ...
...done.
```

之后再次重启服务器：

```
$ ./rabbitmq-server -detached
Activating RabbitMQ plugins ...
6 plugins activated:
* amqp_client-2.7.0
* mochiweb-1.3-rmq2.7.0-git
* rabbitmq_management-2.7.0
* rabbitmq_management_agent-2.7.0
* rabbitmq_mochiweb-2.7.0
* webmachine-1.7.0-rmq2.7.0-hg
```

你可以列出启用的插件，以确保 STOMP 插件禁用成功了：

```
$ ./rabbitmq-plugins list -e
[e] amqp_client                       2.7.0
[e] mochiweb                          1.3-rmq2.7.0-git
[E] rabbitmq_management               2.7.0
[e] rabbitmq_management_agent         2.7.0
[e] rabbitmq_mochiweb                 2.7.0
[e] webmachine                        1.7.0-rmq2.7.0-hg
```

你可以看到 list 选项展示了启用的插件及其版本号，而 STOMP 插件并没有出现在列表中。如果你想学习更多有关 rabbitmq-plugins 命令的话，则可以查阅它的手册：http://www.rabbitmq.com/man/rabbitmq-plugins.1.man.html。

你已经见识到了插件的强大之处，现在该来创建你自己的插件了。准备好了吗？因为在下一节我们就将要使用 Erlang 来编码了。

12.2 制作你自己的插件

到目前为止，我们讨论了插件可以用来做些什么；现在该讨论如何创建你自己的插件了。我们的目标是将自定义交换器添加到 RabbitMQ 中。什么情况下你会需要一个新的交换器类型呢？想象一下，你需要使用 RabbitMQ 来为聊天应用建模，该模型中有一个全局聊天室，所有的用户都连接到这里。每位用户拥有自己的队列，绑定到全局 fanout 交换器上。每次新消息发送到该交换器上时，该消息会群发给所有绑定到此的队列上。现在，如果有一个新的客户端连接到这个聊天室的话，会怎样呢？虽然他们会得到发送给聊天室的所有新消息，但是他们无法了解在加入之前会话的上下文。如果你能将最近 20 条消息发给客户，好让他们能清楚聊天室的情况的话，用户体验就会更好。使用默认的 AMQP 组件是无法做到这一点的。在消息被队列消费之后，我们就再也无法在服务器上看到它了。可以通过创建能够缓存最近路由的 20 条消息的交换器来做到这一点。那样的话，当有新用户连接上聊天室的时候（这意味着新队列绑定到了该交换器上），你的交换器就会将最近的 20 条消息投递过去。图 12.1 描述了这个想法：用虚线隔离出来的那块区域就是交换区需要添加到应用程序中的功能。虽然你并不需要用 RabbitMQ 来实现聊天室，但总有那么一天你的消费者需要知道交换器上最近发生过的事情。实现该功能的简单方式就是这类交换器。

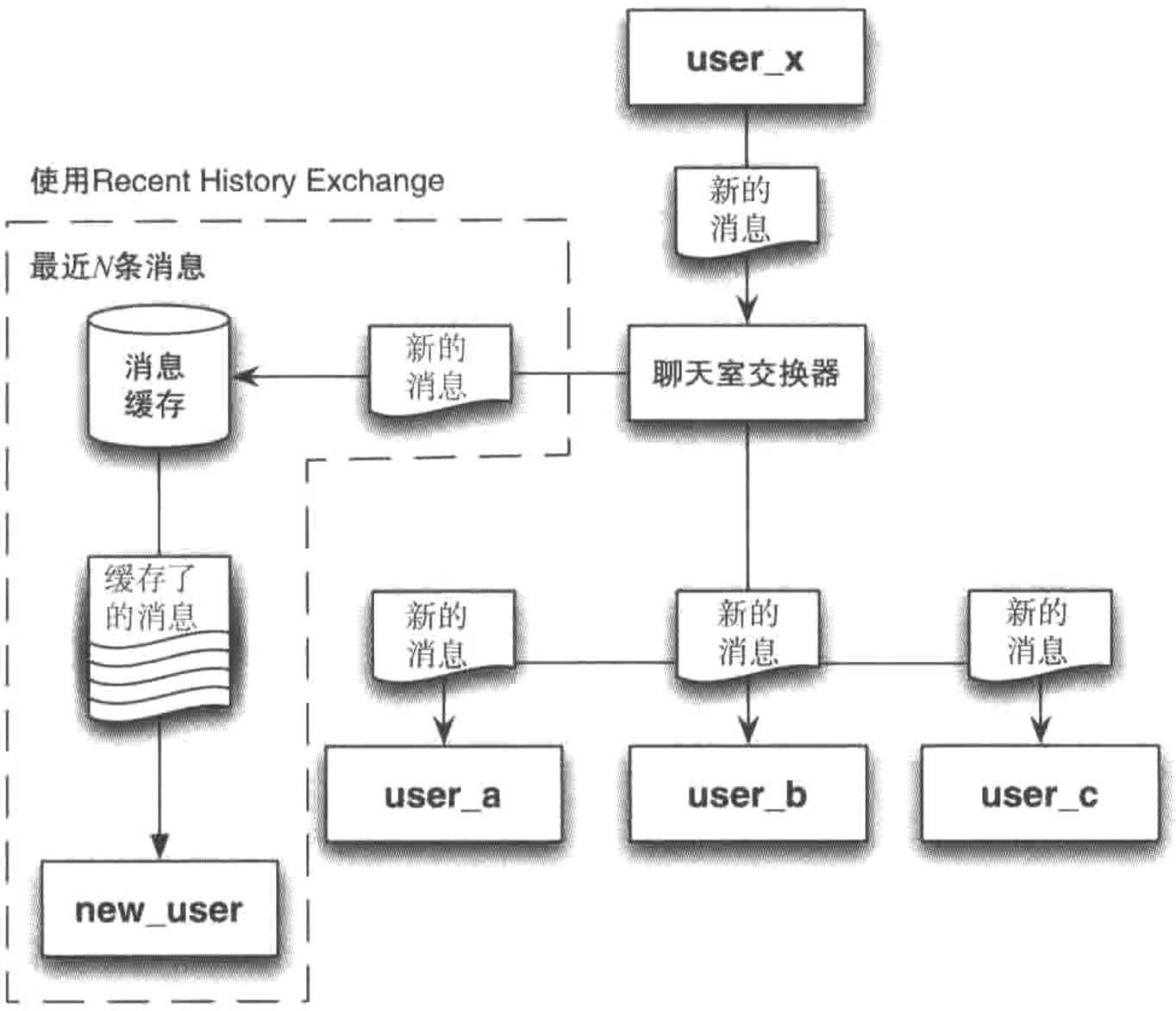

图 12.1 聊天室：使用默认路由并使用自定义交换器

在本节中你将实现该交换器，我们称它为 RabbitMQ Recent History Exchange（RabbitMQ 最近历史交换器）。读者可以参阅该项目的完整源代码，文件夹的名字是 rabbitmq-recent-history-exchange。在下一节中，你将设置开发环境，准备开始创建插件。记住：你将使用 Erlang 来编写。

12.2.1 获取 RabbitMQ Public Umbrella

为了编写自己的插件，你需要先设置基础开发环境。RabbitMQ 开发者已经为你制定了开发环境，即 RabbitMQ Public Umbrella（Rabbit 公伞，以下简称 umbrella）。通过把它从代码库中签出，然后把自己的插件添加到项目结构中就算安装好了。为了获取 umbrella 源代码，你需要先设置 Mercurial。它是一个分布式版本控制系统，和 Git 相似。RabbitMQ 开发人员使用 Mercurial 系统来发布和管理源代码。如果你已经安装了的话，那就直接使用即可。Mercurial 是一个多平台的系统，因此你应该可以在自己的平台上进行安装。整个安装过程很简单；请到 Mercurial 网站上（http://mercurial.selenic.com/）下载适合你电脑的版本。如果之前没有使用过 Mercurial 的话也别担心；你只需其中的一些命令来下载 umbrella 源代码。

如果你是一直按照本书代码示例来的话，那么你应该已经安装好了 Python；否则的话，你也应该安装它，因为 umbrella 下的一些工具需要 Python 的支持（包括 Mercurial 本身）。参阅 Python 网站的安装说明：http://www.python.org/。最后需要的是在本机安装 Erlang；如果你能运行 RabbitMQ 的话，那么你一定已经安装好了。和 Python 一样，请参阅 Erlang 网站来获取安装说明：http://www.erlang.org/。

现在让我们来获取 umbrella 的源代码。假设你安装好了 Mercurial，你可以使用下列命令来获取源代码：

```
$ hg clone http://hg.rabbitmq.com/rabbitmq-public-umbrella/
```

在把库克隆到本机之后，现在就该从源代码中获取项目了。运行下列命令；不过先泡一杯咖啡吧，因为过程有点耗时：

```
$ cd rabbitmq-public-umbrella
$ make co
```

为了检测设置是否正确，你可以运行下列两个命令来编译 rabbitmq-stomp 插件：

```
$ cd rabbitmq-stomp/
$ make
```

如果一切正常的话，你应该看到控制台上输出的结果如下所示：

```
[elided] generate deps
[elided] fix test deps
sed -i -e 's|build/deps.mk|$(DEPS_FILE)|' build/deps.mk
rm -rf build/dep-ezs
mkdir -p build/dep-ezs
....
many lines trimmed out
....
touch dist/.done.0.0.0
touch dist/.done
```

下一步要做的是为你的插件创建文件夹结构。在继续下一节前，我们先进入 umbrella 文件夹中。

12.2.2 设置文件夹结构

你将按照开放电信平台（Open Telecom Platform，OTP）编码标准来构建自己的插件——你将遵循一个特定的文件夹结构、文件命名约定，并采用适合 Erlang 的一些编码模式来组织源代码。由于本书主要讨论的是 RabbitMQ，因此我们不会花费太多时间来讲述 Erlang 的细节；但是如果你很好奇，想要学习更多内容的话，我们建议你读一下由 Manning 出版的 *Erlang and OTP in Action*（http://manning.com/logan/）或者 *Learn You Some Erlang for Great Good*（http://learnyousomeerlang.com/）。即便没有任何 Erlang 经验，你也可以轻松理解示例代码。需要注意的是，如果你之前从未使用过 Erlang 或者函数式编程的话，一些技术可能看起来很陌生。

插件的文件夹结构十分简单。由于你称它为 RabbitMQ Recent History Exchange，因此在 umbrella 项目内创建一个叫作 `rabbitmq-recent-history-exchange` 的文件夹。在该文件夹下创建 `src` 文件夹来存放你的源代码。就是这么简单：

```
cd ..
mkdir rabbitmq-recent-history-exchange
cd rabbitmq-recent-history-exchange
mkdir src
```

下一步是将 umbrella 构建系统包含到你的项目里面。

12.2.3 包含插件构建系统

为了能够使用 umbrella 构建系统，你需要将一些文件添加到项目根文件夹下。第一个文件是 Makefile，它引用了 umberlla 的 make 文件。添加以下内容。

```
include ../umbrella.mk
```

通过引用 umbrella.mk 文件，你将受益于那些已有的命令。这些命令可以用来将你的插件打包为 .ez 文件；另一些命令则能从项目文件夹下在服务器内部直接运行你的插件，这使得测试起来很简单。还有些命令则是用来清理构建文件的。请参考 umbrella 文件夹下的 README.makefiles 以获取完整的选项列表。

现在 Makefile 已经就位了，你可以添加 package.mk 文件，它包含了构建系统所需的插件具体配置选项。在你的项目文件夹下创建该文件，并添加以下内容：

```
DEPS:=rabbitmq-server rabbitmq-erlang-client
RETAIN_ORIGINAL_VERSION:=true
```

看起来就是这么简单。你做的是在 DEPS 宏中指明了插件依赖，并且将 RETAIN_ORIGINAL_VERSION 宏设置为 true，让构建系统获取你将要赋予给插件的版本号。由于你准备构建自定义交换器，你需要 rabbitmq-server 项目中一些已有的功能和 rabbitmq-erlang-client 的一些特性；因此，你在 package.mk 文件中指明了该插件依赖于这两个项目。umbrella 构建系统会负责为你解析依赖关系，是不是很棒？现在你已经准备就绪开始编码你的插件了，那么就让我们编写应用规范文件吧。

12.2.4 创建 Erlang 应用文件

现在基本环境已经设置就绪，让我们创建应用规范文件吧。你也许会奇怪那是什么——应用程序会是什么样子呢？ Erlang 程序是由应用文件构成的，而每个应用文件包含了具体实现功能的模块。像 Erlang 模块列表，包括应用程序或者配置选项，都是在应用规范（application specification）文件中指定的。当创建应用规范文件时，你需要指明 Erlang 模块列表，这看起来有点乏味。幸运的是该步骤可以通过使用应用规范模板（application specification templates）来简化，umbrella 构建系统会在稍后自动填充这些信息。这意味着你无须担心如何保持模块列表的更新；构建系统会负责这些工作。那么虽然通常在 Erlang 项目中，你将创建普通的应用程序文件，但是在本例中，你将为其创建一个模板，并让 umbrella 系统来自动填写信息。

让我们在 src 文件夹下创建该模板文件，并命名为 rabbitmq_recent_history_exchange.app.src。该文件包含以下内容：

```
{application, rabbitmq_recent_history_exchange,
 [{description, "RabbitMQ Recent History Exchange"},
  {vsn, "0.1.0"},
  {modules, []},
  {registered, []},
  {applications, [kernel, stdlib, rabbit, mnesia]}]}.
```

这里你看到的是一个 Erlang 元组（tuple），它是一个复合数据类型，拥有固定数量的 Erlang 值（数值）[3]。如果对该结构进行简化，你可以看到它具有如下的形态：

```
{application, application_name, [{key1, val1}, ..., {keyN, valN}]}.
```

元组中的第一个元素是名为 application 的原子类型——原子类型就像 Ruby 中的符号（symbol）或者 Clojure 中的关键字（keyword）。第二个元素是用于保存应用名称的原子类型。最后，第三个元素是一个 Erlang 列表，以字符 [和] 进行分隔。这是一个属性列表（property list），因为它存放了键和值的元组。如果你仔细查看了内容的话，就会发现有一个 description 的属性 {description, "RabbitMQ Recent History Exchange"} 描述了该应用是用来做什么的。你可以用任何字符串来作为该应用的描述。vsn 值则指明了应用的版本号。由于应用才刚刚创建，因此你把它标记为 *0.1.0*。之后是应用程序需要使用的 Erlang 模块列表。我们之前提到过，该列表会被 umbrella 构建系统自动填充。registered 属性代表应用程序会注册哪些进程，该示例中为空列表。最后，你指明了哪些应用必须运行才能让插件正常工作。Erlang kernel 和标准库 stdlib 是必需的。你依赖于 RabbitMQ 服务器的正常运行，因此添加了 rabbit 应用。最后一点，因为你将使用 Erlang Mnesia 数据库，所以你把它也加入进来。

在开始编写交换器逻辑之前，让我们先确保 Makefiile 设置正确。在插件文件夹下输入 make。你应该可以在命令行上看到许多输出。这很正常：构建系统会将所有插件的依赖如 Erlang AMQP 客户端和服务器自身等全部 make 起来。在 make 过程结束后，你会注意到在项目根目录下有一些新的文件夹。最有意思的是那个叫

3 如果你想了解更多有关Erlang数据类型的话，请参阅http://www.erlang.org/doc/reference_manual/data_types.html。

作 dist（distribution）的文件夹。这里存放着最终生成的插件文件。在终端输入下列命令：

```
$ ls dist
amqp_client-0.0.0.ez rabbit_common-0.0.0.ez
rabbitmq_recent_history_exchange-0.1.0-rmq.ez
```

这里你看到了插件文件和它的依赖。虽然插件没有实现功能，但是你至少可以测试构建系统是否设置正确。在完成插件的最终版本之后，你需要将这些文件复制到 RabbitMQ 安装目录下的 plugins 文件夹下。现在让我们开始编写 Erlang 代码吧。

12.3 创建自定义交换器模块

Erlang 源代码是以模块的方式组织起来的。模块内的函数实现了应用程序需要提供的功能。不像 Java 或者 C#，Erlang 里没有类或者包，这让结构变得扁平并且简单。插件只需要一个模块来包含自定义交换器的实现。你将在 src 文件夹下创建名为 rabbit_exchange_type_recent_history.erl 的文件，并随着本节内容的深入，你会往其中添加更多的内容。

你怎样了解交换器的内部情况呢？ RabbitMQ 如何知道应该调用模块里的哪些函数来完成消息的路由、队列绑定等呢？在面向对象编程（OOP）中，你拥有接口（interface）的概念。你可以在接口中定义一系列的方法，实现该接口的类必须实现这些方法，就像签订了合约一样。举例来说，为了实现访问者（Visitor）模式，你可以查看 Visitor 接口，它会告诉你需要提供一个 Visit 方法；Iterator 会实现 hasNext 和 next 方法，等等。在 Erlang 中有相同的概念，但是不同的名称：behaviour（行为）。注意这个单词的英式拼写。

Erlang behaviour 确定了模块需要实现和导出哪些函数，因此调用该模块的代码才知道该如何使用它。另一个与 OOP 世界不同的地方在于，Erlang 没有 Java 那样可见性的概念；举例来说，Erlang 没有 public、protected、private 修饰符。你拥有的就是一个模块导出列表。因此，如果模块实现了函数 foo、bar 和 baz，但只导出 foo 的话，那么 bar 和 baz 将无法从外界调用。除了模块的函数导出这块内容

之外，你还需要知道函数可以接收的参数个数。Erlang 有函数参数数量（function arity）的概念——一个函数接收的参数数量。Erlang 可以接受名字相同，而参数数量不同的函数；因此 behaviour 可以确定函数 `foo/1` 和 `foo/2` 是两个不同的函数。当指定函数名称时，将一个正斜杠放在函数名称和其参数之间。现在让我们回来继续讨论 behaviour：你如何知道交换器的工作情况呢？ RabbitMQ 暴露了一个交换器 behaviour，名为 `rabbit_exchange_type`，它会告诉你需要实现哪些函数来成为一个符合要求的交换器。由于你的交换器是一个加强版的 fanout 类型交换器，缓存最近 20 条消息，因此你将基于 RabbitMQ 上真实的 fanout 交换器来进行实现，这样会简化任务。

看了图 12.2，你就明白需要完成一个和默认 fanout 交换器不同的实现。首先，你需要缓存已路由的消息。每当交换器路由消息的时候，你就将该消息存储到某个数据库中。该功能会在函数 `route/2` 中实现。然后，当队列绑定到你的交换器上的时候，你需要把你缓存的那些消息投递过去。该功能会在函数 `add_binding/3` 中实现。最后，当你的交换器删除的时候，你需要丢弃缓存以防内存泄露，这会在函数 `delete/3` 中处理。注意，同时你也需要实现其他功能。在图 12.2 中提到的这些是与 fanout 交换器默认实现之间的差异。最后一个实现细节是，你将要使用 Mnesia（Erlang 的内建数据库）。RabbitMQ 就是用它来存储绑定和交换器元数据信息的。你会一步步地来实现该交换器，并在最后，我们将向你展示模块的完整源代码。

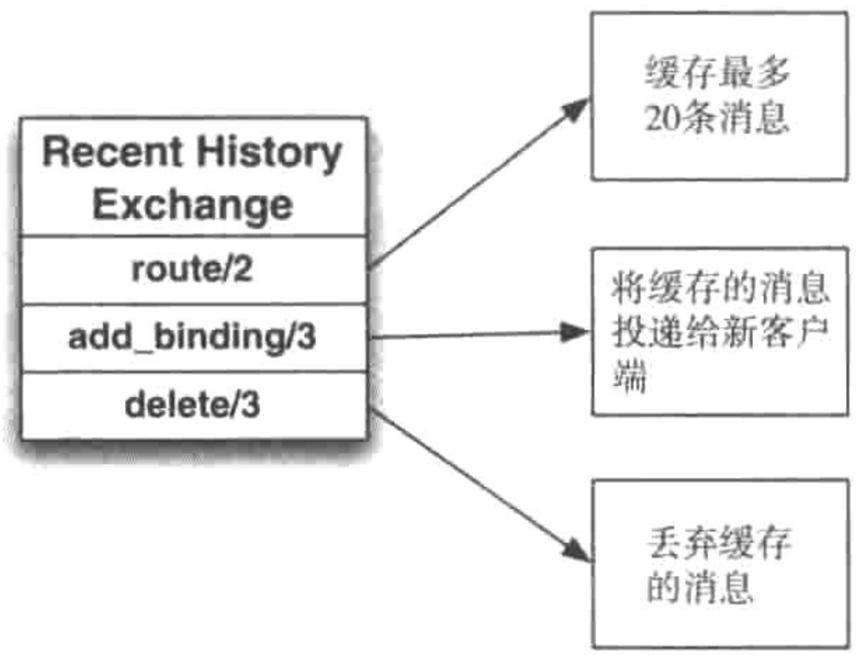

图 12.2 Recent History Exchange 的详细功能说明

现在让我们在 `src` 文件夹下创建 `rabbit_exchange_type_recent_history.erl`，并添加以下内容。

清单 12.1 rabbit_exchange_type_recent_history.erl

```
-module(rabbit_exchange_type_recent_history).
-include_lib("rabbit_common/include/rabbit.hrl").
-include_lib("rabbit_common/include/rabbit_exchange_type_spec.hrl").

-behaviour(rabbit_exchange_type).

-export([description/0, serialise_events/0, route/2]).
-export([validate/1, create/2, delete/3, add_binding/3,
         remove_bindings/3, assert_args_equivalence/2]).
-export([setup_schema/0]).
```

❶ 模块定义

❷ 导入需要的头

❸ 指明 behaviour

❹ 模块中导出的函数

首先，你给出了模块名称❶，它必须与去掉 .erl 扩展名的文件名相匹配。然后，你包含了一些模块需要的库❷，并定义模块实现了 rabbit_exchange_type behaviour ❸。最后，你声明了模块导出的函数列表❹。注意，这段代码中有多个 export 声明。这么做只是为了可读性，因为你完全可以将所有的函数声明放到一个 export 语句中去。每个 export 声明都有用方括号 [] 分隔的 Erlang 列表，列表是以逗号分隔的、形如函数名 / 元数(*function_name/arity*)的函数。举例来说，你需要实现一个叫作 description 的函数，不携带任何参数；一个叫作 route 的函数，携带两个参数；等等。前面的两个 export 声明是 rabbit_exchange_type behaviour 要求的，而最后一个包含 setup_schema 函数的 export 声明则只和该模块有关。如你所见，除了你需要实现的 behaviour 之外，你也可以导出更多的函数。现在让我们继续看看剩下的代码。

12.3.1 将交换器注册到 RabbitMQ

为了能让 RabbitMQ 加载并使用交换器，我们必须让它知道交换器在哪儿。RabbitMQ 维护了一个注册表，用这张表来跟踪所有的交换器类型及其模块名称。假设你将消息发布到 fanout 交换器上。RabbitMQ 会进入注册表，检查是由哪个模块来实现 fanout 交换器的。当它找到模块名称之后，就会继续调用该模块上的路由函数。在我们的案例中，你需要找到一个方法将你的交换器添加到注册表中，这样服务器才能找到它。RabbitMQ 支持启动步骤（boot step）的概念——当服务器启动的时候会调用一系列的步骤。你将在模块中添加一个启动步骤，然后 RabbitMQ 会魔法般地执行该步骤，并将你的交换器添加到 rabbit_registry 中。将下列代码添加到模块中，以完成上述功能。

清单 12.2 rabbit_exchange_type_recent_history.erl

```
-rabbit_boot_step({rabbit_exchange_type_rh_registry,
[{description, "recent history exchange type: registry"},
  {mfa, {rabbit_registry, register,
          [exchange, <<"x-recent-history">>,
           ?MODULE]}},
  {requires, rabbit_registry},
  {enables, kernel_ready}]}).

-rabbit_boot_step({rabbit_exchange_type_rh_mnesia,
  [{description, "recent history exchange type: mnesia"},
    {mfa, {?MODULE, setup_schema, []}},
    {requires, database},
    {enables, external_infrastructure}]}).

-define(KEEP_NB, 20).
-define(RH_TABLE, rh_exchange_table).
-record(cached, {key, content}).
```

❶ 注册交换器

❷ 初始化数据库

❸ 定义常量和记录

rabbit_boot_step ❶包含如下模块：首先，description 用来表明启动步骤是做什么的（注意它就是作为文档说明用的；在这里你可以填写任何内容，只要是字符串即可）。下一部分是 mfa，即模块函数参数部分（module function arguments），在这里你调用了 rabbit_registry 中的函数 register，将交换器类型和相关模块作为参数传入。在 AMQP 层面，你的交换器类型被命名为 x-recent-history。根据 AMQP 规范，你添加到协议的所有自定义元素都需要添加 x- 前缀。最后是 requires 和 enables 部分，在这里你告诉 RabbitMQ 为了注册该交换器，rabbit_registry 必须先运行起来（这很明显）；同时，你的交换器对于 kernel_ready 事件的触发来说是必要条件。依赖于 kernel_ready 事件的启动步骤将必须等待你的交换器初始化完成才行[4]。

同时，你利用 rabbit_boot_step 系统来初始化自己的插件❷。由于插件将使用 Mnesia 来存储缓存的消息，因此你添加了一个额外的步骤来为插件设置数据库结构；启动步骤中的 mfa 部分会调用属于你的交换器模块的 setup_schema 函数。你看到的 ?MODULE 片段和 C 语言中的宏相似，所以在该案例中它会展开为你的模块名称。由于 setup_schema 函数不需要任何参数，因此此处提供一个空列表 []。

最后，你定义了两个宏：KEEP_NB 和 RH_TABLE ❸，它们分别代表了你想要保

4 如果你想要学习更多关于RabbitMQ启动过程的话，则可以参考下列文档：https://github.com/videlalvaro/rabbit-internals/blob/master/rabbit_boot_process.md.。

存在缓存中的消息数目以及你将要创建的 Mnesia 表的名称。说到 Mnesia 表，你需要定义表的结构，在代码最后用 -record 来完成。你存储的记录称为 cached，这些记录拥有两个元素：key 和 content。key 属性用来存储交换器名称，content 属性用来存储通过该交换器的最近 20 条消息。这样就建立了从交换器名称到缓存消息之间易于访问的映射。当需要从缓存中投递近 20 条消息时，你需要通过交换器名称来查找消息列表。现在让我们来看看 setup_schema/0 函数。

清单 12.3 rabbit_exchange_type_recent_history.erl

```
setup_schema() ->                                          ❶ 函数定义
  case mnesia:create_table(?RH_TABLE,                      ❷ 创建表
          [{attributes, record_info(fields, cached)},
           {record_name, cached},
           {type, set}]) of
      {atomic, ok} -> ok;                                  ❸ 创建成功
      {aborted, {already_exists, ?RH_TABLE}} -> ok         ❹ 创建失败
 end.
```

这段函数易于理解。首先是函数头❶，包括函数名称、括号和括号之间的参数列表（该示例中没有参数），以及箭头 ->（用来指示之后的内容就是函数体了）。这里有一个下列形式的 case 表达式：

```
case Expr of
 Pattern1 ->
  Body1;
 ...;
 PatternN ->
  BodyN
end
```

Erlang 会对 Expr 进行求值，同时根据表达式的结果执行 Body 部分；最后用 end 单词来结束 case 表达式。如你所见，这和其他语言中的 switch/case 语句很相似。在这里，Expr 为对 mnesia:create_table 的调用❷。让我们来看看细节：

```
mnesia:create_table(?RH_TABLE,
        [{attributes, record_info(fields, cached)},
         {record_name, cached},
         {type, set}])
```

此处你创建了名为 rh_exchange_table 的表，因为 ?RH_TABLE 宏会展开成为表名。attributes，或者说是表的 columns，将会使用 cached 记录中的域——就

是你先前定义的 key 和 content 域。你告诉 Mnesia 记录的名称是 cached，表类型为 set，该类型的表不会有重复的值。如果使用相同的键来存储新的值，那么旧的值会被新的值覆盖。

回到 setup_schema 函数，如果 Expr 的结果是 {atomic, ok} ❸，那就意味着表创建成功了，返回 ok。注意在 Erlang 中，和其他函数式编程语言一样，没有 return 关键字。函数会隐含地返回最后一条表达式的结果。现在，如果 mnesia:create_table 返回 {aborted, {already_exists, ?RH_TABLE}} ❹，也将返回 ok。这是因为表已经存在于 Mnesia 中了，很有可能是之前 RabbitMQ 启动的时候创建的，所以也就无须再做什么了。

12.3.2 实现交换器 behaviour

现在该来实现交换器 behaviour 了。由于你将重用 RabbitMQ 提供的默认实现，因此你将添加的一些函数实现都很简单。让我们把下列函数添加到模块当中去。

清单 12.4 rabbit_exchange_type_recent_history.erl

```
description() ->                                          ❶ 交换器描述
  [{name, <<"recent-history">>},
   {description, <<"List of Last-value caches exchange.">>}].

serialise_events() -> false.                              ❷ 绑定事件序列化

remove_bindings(_Tx, _X, _Bs) -> ok.                      ❸ 其他函数
validate(_X) -> ok.
create(_Tx, _X) -> ok.

assert_args_equivalence(X, Args) ->                       ❹ 交换器参数等价
  rabbit_exchange:assert_args_equivalence(X, Args).
```

首先，description ❶函数仅仅用于提供信息。它返回了一个属性列表，包含了交换器名称和它的描述。很简单吧！下一个函数是 serialise_events ❷，RabbitMQ 使用该函数来决定是否序列化交换器接受的绑定事件。在该示例中，你不需要这样的功能，因此返回 false。remove_bindings、validate 和 create 的实现❸是非常直白的。因为这些针对交换器的操作发生时，你不需要做任何日志方面的事情，所以此处返回原子 ok；RabbitMQ 会执行默认的动作。最后，你将对 assert_args_equivalence 的调用代理给了 RabbitMQ 的 rabbit_exchange 模

块的实现❹。你无须了解这些函数的细节。不过如果你想知道更多内容的话，请参考服务器源代码中的 rabbit_exchange_type.erl 文件，它对每个 behaviour 函数都有解释。

在你完成了这些基础设置之后，现在该来定义实现交换器功能的函数了。你需要实现函数 route/2、delete/3 和 add_binding/3。让我们从 route/2 的代码开始。

清单 12.5 消息路由——rabbit_exchange_type_recent_history.erl

```
route(#exchange{name = XName},                          <-- ❶ 抽取交换器名称
      #delivery{message = #basic_message{
                            content = Content             <-- ❷ 抽取消息内容
                          }}) ->
cache_msg(XName, Content),                              <-- ❸ 缓存消息
rabbit_router:match_routing_key(XName, ['_']).          <-- ❹ 路由消息
```

顾名思义，route 接收两个参数：交换器记录和需要路由的消息。Erlang 中的记录（record）和 C 语言中的结构类似：它是一个将域映射到值的数据结构。记录的一个好处是你可以通过域的名称来访问它们的值。Erlang 有一个叫作模式匹配（pattern matching）的技术，可以用于抽取数据结构中的消息片段。这种做法简化了数据访问，同时不需要临时变量。在该示例中，你只对交换器的名称感兴趣，因此虽然函数持有一条完整的交换器信息记录，但是通过将代码 #exchange{name = XName} 放置在函数头上，你就能抽取名称的值，并将其绑定（bind）到变量 XName 上❶。于是该变量就拥有了交换器名称的值，你就可以在剩下的函数体中使用该变量了。一旦你掌握了这种 Erlang 技术，你就能明白它是如何简化代码的。你对函数中的第二个参数也使用相同的方式来抽取消息内容❷，并将值绑定到变量 Content 上。该变量将获得消息投递内容——AMQP 消息属性和有效载荷。

函数体非常简单。首先，你调用了函数 cache_msg/2，向其传递了交换器名称和消息内容作为参数❸。你一会儿就会看到函数代码。一旦消息缓存之后，你调用默认的 rabbit_router ❹来实现和 fanout 交换器一样的行为。下一步是为 cache_msg/2 和辅助函数 store_msg/2 编写代码。将下列代码添加到模块文件中。

清单 12.6 rabbit_exchange_type_recent_history.erl

```
cache_msg(XName, Content) ->
  rabbit_misc:execute_mnesia_transaction(
    fun () ->
      Cached = get_msgs_from_cache(XName),
      store_msg(XName, Cached, Content)
    end).

store_msg(Key, Cached, Content) ->
  mnesia:write(?RH_TABLE,
    #cached{key = Key,
        content = [Content|lists:sublist(Cached, ?KEEP_NB-1)]},
    write).
```

匿名函数作为参数

获取缓存消息

存储消息

添加消息到列表中

第一个函数 cache_msg/2 接收两个参数:交换器名称和消息内容。在该函数中，你需要访问 Mnesia 以获取消息，消息可能在缓存之中，同时将最新的消息添加上去。由于你的交换器可能会并发调用，因此你需要将读取和更新的操作运行在一个事务当中以确保数据一致性。RabbitMQ 提供了一个帮助函数 rabbit_misc:execute_mnesia_transaction/1 来做到这一点。该函数接收另一个函数作为参数，并将该函数运行在 Mnesia 事务的上下文中。就像许多函数式编程语言一样，Erlang 提供了定义匿名函数的方法，在 Erlang 世界中它们被称为 funs。函数是“一等公民”，这意味着它们可以被当作普通值来传递给其他函数，同时也可以被其他函数返回。如果你使用过 JavaScript 的话，那么在使用回调函数时你很可能使用过一些匿名函数了。Erlang fun 的简化语法如下所示：

```
fun(Arg1, Arg2, ...., ArgN) ->
    Expr1,
    Expr2
    ....,
    ExprN
end
```

该代码片段中的 fun 不接收任何参数，这是由于它封闭了传递给 cache_msg 函数的参数，形成了闭包。该 fun 首先做的事是通过调用帮助函数 get_msgs_from_cache/1，传入交换器名称作为参数，来获取缓存消息。之后将函数调用结果绑定到变量 Cached 上，然后传递给函数 store_msg/3，由它来负责存储数据。通过这两个分隔的步骤，即从 Mnesia 获取数据和将数据存储到 Mnesia，你就能重用这些代码了。

函数 store_msg/3 同样简单。它调用了 mnesia 模块的函数 write/3。第一个参数是表名；第二个参数是想要存储的记录。注意，你使用交换器名称来作为 key 字段的值。为了存储缓存内容，你使用语法 [Head|Tail] 来动态创建 Erlang 列表。列表的头是新缓存的元素，列表尾是已缓存的 ?KEEP_NB - 1 个元素的子列表。通过这种做法，你能够确保最多缓存 20 个元素。你可能好奇为什么要将新元素添加到列表头。Erlang 当中的列表是以链表实现的，因此将元素添加到列表头，并在投递消息时，将列表反转成它们到达时的顺序，这样做并不花费多少性能。Mnesia 函数的最后一个参数是 write，它用来向 Mnesia 请求一把 ?RH_TABLE 表的写锁（lock）。现在让我们看看 get_msgs_from_cache/1 函数的代码。将下列代码添加到模块文件中去。

清单 12.7 rabbit_exchange_type_recent_history.erl

```
get_msgs_from_cache(XName) ->
  rabbit_misc:execute_mnesia_transaction(
    fun () ->
      case mnesia:read(?RH_TABLE, XName) of
        [] ->
          []; ❶ 返回一个空的列表
        [#cached{key = XName, content=Cached}] ->
          Cached ❷ 返回缓存的内容
      end
  end).
```

该函数的代码也十分简单。你再一次使用 execute_mnesia_transaction/1 函数，传递了含有 case 表达式的 fun 作为参数。case 表达式会提供表名和交换器名来调用 mnesia:read/2。如果数据库返回以 [] 表示的空列表的话，你就返回空的列表❶。如果你得到了以 #cached 记录形式的列表，就能从该记录中抽取交换器名称（exchange name）和内容（content）。现在你可能好奇：为什么你要将 key 字段的值和变量 XName 关联起来。这又和 Erlang 中的模式匹配有关。在该示例中，你使用这种技术来做逻辑检查，以确保你获取的值的确是关联到交换器名称的。它是如何工作的呢？当函数被调用时，变量 XName（也是该函数唯一的参数）将会绑定传递给函数的值。在 Erlang 中，与许多函数式编程语言一样，变量不会发生变化[5]。在这里的变量和高中数学中的变量工作的方式一样。绑定到 XName 的值在函数

5 在Erlang中，变量只能被赋值一次。——译者注

执行的范围和生命周期中都无法更改。因此，在 case 语句的第二个表达式中，它唯一完成匹配的情况必须是 key 字段的值和 XName 变量的值相匹配。这样做就能确保你从 Mnesia 取回的值就是当前交换器缓存的值。尽管最初看起来有点复杂，但它会在之后的编程中简化你的代码；你再也不用在代码中添加无止境的 if/then/else 语句了，因为你将预先对变量使用模式匹配（pattern match）。如果模式匹配失败的话，那么代码将不会执行。回到代码中来，你可以看到 case 表达式的第二部分返回缓存的内容❷。

get_msgs_from_cache/1 描述了交换器用于路由和缓存消息的代码。现在该来看看当某人删除其中一个自定义交换器时该怎么办。下列代码实现了 delete/3 函数。让我们把它添加到模块中去。

清单 12.8 rabbit_exchange_type_recent_history.erl

```
delete(_Tx, #exchange{ name = XName }, _Bs) ->
  rabbit_misc:execute_mnesia_transaction(
    fun() ->

      mnesia:delete(?RH_TABLE, XName, write)
    end),
  ok.
```

delete/1 函数也很简单。每当交换器被删除的时候，RabbitMQ 就会调用该方法。虽然在示例中该方法接收了三个参数，但是你仅使用第二个参数来抽取交换器名称的值，之后将使用该值从 Mnesia 中删除属于该交换器的信息。记住，你的自定义交换器类型可能会有多个实例，每个实例缓存了不同的消息。为了防止内存泄漏，每当交换器类型的示例从服务器中移除时，你需要负责将与之关联的消息一并删除。下一个要实现的回调函数的功能是将队列绑定到交换器上。将下列代码添加到源文件中去。

清单 12.9 rabbit_exchange_type_recent_history.erl

```
add_binding(_Tx, #exchange{ name = XName },
            #binding{ destination = QName }) ->

  case rabbit_amqqueue:lookup(QName) of
    {error, not_found} ->

      queue_not_found_error(QName);
    {ok, #amqqueue{ pid = QPid }} ->
```

❶ 获取队列Pid

```
    Cached = get_msgs_from_cache(XName),
    Msgs = msgs_from_content(XName, Cached),
    deliver_messages(QPid, Msgs)
end,
ok.
```

❷ 转换消息

每当交换器添加一个绑定时，函数 add_binding/3 就会被 RabbitMQ 调用，让你可以对绑定做些额外的操作。对于该示例来说，一个新的绑定意味着一个新的客户端连接到该交换器上，因此，这时就该将 Mnesia 缓存的消息投递给客户端了。从本质上来讲，之前所有的代码都是为了支持该函数。为了将消息投递给队列，你需要队列的 Pid，即进程 ID（process ID）。Pid 就像是队列的地址，你可以将消息（是 Erlang 消息，而非 AMQP 消息）发送到该地址。你可以使用队列 Pid，通过将消息委托给队列模块来投递给消费者。记住在 RabbitMQ 中，消费者订阅的是队列，因此在交换器层面是没有消费者的概念的。为了获取队列 Pid，你调用帮助函数 rabbit_amqqueue:lookup/1，传入队列名称作为参数❶，并且要么返回 Pid，要么返回包含原子 {error, not_found} 的元组。如果 RabbitMQ 无法找到你需要的队列的话，那么通过调用函数 queue_not_found_error/1 来返回一个协议错误。如果你得到了 QPid 的话，那么获取缓存中的消息内容，并通过调用函数 msgs_from_content/2 将内容转换成真正的 AMQP 消息，以 AMQP 消息列表的形式返回。最后，你将这些消息传递给函数 deliver_messages/2，由它来迭代消息列表并将每一条消息投递到目的队列。在处理完消息投递之后，你将向服务器返回原子 ok，以表示绑定操作过程成功了。

让我们看一下之前代码中使用的帮助函数，以对整个实现有充分的了解。

清单 12.10 rabbit_exchange_type_recent_history.erl

```
queue_not_found_error(QName) ->
  rabbit_misc:protocol_error(
    internal_error,
    "could not find queue '~s'",
    [QName]).

msgs_from_content(XName, Cached) ->
```

❶ 协议错误

```
        lists:map(
          fun(Content) ->
              {Props, Payload} = rabbit_basic:from_content(Content),
              rabbit_basic:message(XName, <<"">>, Props, Payload)
          end, Cached).

deliver_messages(Queue, Msgs) ->
  lists:map(
    fun (Msg) ->
      Delivery = rabbit_basic:delivery(false, false, Msg, undefined),
      rabbit_amqqueue:deliver(Queue, Delivery)
  end,  lists:reverse(Msgs)).
```

❸ 创建一条AMQP消息

❷ 创建消息属性和内容

❹ 将消息打包成一条“投递（delivery）”

❺ 将消息投递到队列

首先是一个简单的帮助函数，用来向客户端发送协议错误❶。它接收队列名称作为参数，并使用该参数来格式化字符串，告诉用户无法找到队列。然后是函数 `msgs_from_content/2`，它从接收到的内容中生成 AMQP 消息列表。记住，你传递给该函数的是 `Content` 列表；因此你必须对该列表进行映射，对列表中的每个元素应用函数 `fun`。当你调用 map 函数时，必须记得反转消息列表，调整为消息接收时的顺序。为了反转列表，你调用了 Erlang 函数 `lists:reverse/1`。

一条 AMQP 消息是由属性的集合以及有效载荷组成的。你通过调用 `rabbit_basic:from_content/1` 函数将消息从缓存内容中抽取出来❷。该函数返回由消息载荷和属性组成的元组。下一步要做的是将交换器的名称和用来路由该消息的路由键添加到消息中去。在该示例中为空白路由键，由空的二进制数据 <<"">> 表示[6]。为了完成这个操作，你需要调用函数 `rabbit_basic: message/4` ❸，并传递消息属性和载荷，以获得一条完整的 AMQP 消息。

最后一点同样重要，帮助函数 `deliver_messages/2` 接收队列 `Pid` 和消息列表作为参数，并对列表做映射，将每条消息投递给队列。函数首先要做的是通过调用函数 `rabbit_basic:delivery/4` 将消息包装为一条投递（delivery）❹。该函数接收四个参数：第一个参数说明了消息投递是否必须，这里设置为 false；第二个参数说明该消息是非紧急的；第三个参数就是实际的消息；最后一个参数是消息序列 ID，在这里未定义。在创建了消息投递之后，你最终通过调用函数 `rabbit_amqqueue:deliver/2` 将其发送给队列。该函数接收队列 `Pid` 和 `Delivery` 作为参数❺。

---

6 在Erlang中，二进制是展现字符串数据的一种高效的方式。更多有关二进制和其他Erlang数据类型的信息，请参阅*Learn You Some Erlang for Great Good*一书：http://learnyousomeerlang.com/starting-out-for-real#bit-syntax:。

12.3.3 编译自定义交换器

以上代码实现了自定义交换器。如你所见，扩展 RabbitMQ 并添加新的交换器类型是很容易的事。只需要遵循你想要实现的 Erlang behaviour 强制的几条规则即可。为了简化实现，你甚至可以基于 fanout 类型交换器来实现自己的交换器，事情变得更简单了。让我们回顾一下你所做的一切：你重写了消息路由、队列绑定和交换器删除，以便缓存并投递消息。为了让代码干净整洁并让函数简短，在需要写入缓存或者从缓存中读取消息的时候，你编写了一些帮助函数来访问 Mnesia。

现在让我们尝试编译代码，并让 RabbitMQ 与自定义交换器插件一同运行。幸运的是，umbrella 构建系统包含了一个命令，它通过自动安装插件的方式，允许你将插件和服务器合并运行。在插件文件夹下运行下列命令：

```
$ make run-in-broker
```

当按下 Enter 键时，你将看到一大堆的输出。首先你的插件会被编译；并且，如果一切正常的话，RabbitMQ 就会启动。你应该能够看到如下这条消息，表示你的插件已经启用：

```
Activating RabbitMQ plugins ...
2 plugins activated:
* amqp_client-0.0.0
* rabbitmq_recent_history_exchange-0.1.0-rmq
```

像往常一样，RabbitMQ 的 logo 会展现出来，并在 rabbit registry 启动之后，你应该能看到如下输出：

```
external infrastructure ready
starting plugin registry                                    ...done
starting auth mechanism cr-demo                             ...done
starting auth mechanism amqplain                            ...done
starting auth mechanism plain                               ...done
starting statistics event manager                           ...done
starting logging server                                     ...done
starting exchange type direct                               ...done
starting exchange type fanout                               ...done
starting exchange type headers                              ...done
starting recent history exchange type: registry             ...done
starting exchange type topic                                ...done
```

注意到这一行内容：`starting recent history exchange type:registry...done`。这说明你的插件无缝集成到了服务器上。最后，你将留在 Erlang 命令行。暂

时先别输入任何东西。之后你会看到如何关闭并退出 Erlang。在你编写消费者和发布者之前，让我们来看看完整的自定义交换器模块的代码清单。

清单 12.11 rabbit_exchange_type_recent_history.erl

```
-module(rabbit_exchange_type_recent_history).
-include_lib("rabbit_common/include/rabbit.hrl").
-include_lib("rabbit_common/include/rabbit_exchange_type_spec.hrl").

-behaviour(rabbit_exchange_type).

-export([description/0, serialise_events/0, route/2]).
-export([validate/1, create/2, delete/3, add_binding/3,
         remove_bindings/3, assert_args_equivalence/2]).
-export([setup_schema/0]).

-rabbit_boot_step({rabbit_exchange_type_rh_registry,
[{description, "recent history exchange type: registry"},
  {mfa, {rabbit_registry, register,
          [exchange, <<"x-recent-history">>,
           ?MODULE]}},
  {requires, rabbit_registry},
  {enables, kernel_ready}]}).

-rabbit_boot_step({rabbit_exchange_type_rh_mnesia,
  [{description, "recent history exchange type: mnesia"},
    {mfa, {?MODULE, setup_schema, []}},
    {requires, database},
    {enables, external_infrastructure}]}).

-define(KEEP_NB, 20).
-define(RH_TABLE, rh_exchange_table).
-record(cached, {key, content}).

description() ->
  [{name, <<"recent-history">>},
   {description, <<"List of Last-value caches exchange.">>}].

serialise_events() -> false.

route(#exchange{name = XName},
      #delivery{message = #basic_message{
                            content = Content
                           }}) ->
  cache_msg(XName, Content),
  rabbit_router:match_routing_key(XName, ['_']).

validate(_X) -> ok.
create(_Tx, _X) -> ok.

delete(_Tx, #exchange{ name = XName }, _Bs) ->
  rabbit_misc:execute_mnesia_transaction(
    fun() ->
      mnesia:delete(?RH_TABLE, XName, write)
```

```
    end),
  ok.

add_binding(_Tx, #exchange{ name = XName },
            #binding{ destination = QName }) ->
  case rabbit_amqqueue:lookup(QName) of
    {error, not_found} ->
      queue_not_found_error(QName);

    {ok, #amqqueue{ pid = QPid }} ->
      Cached = get_msgs_from_cache(XName),
      Msgs = msgs_from_content(XName, Cached),
      deliver_messages(QPid, Msgs)
  end,
  ok.

remove_bindings(_Tx, _X, _Bs) -> ok.

assert_args_equivalence(X, Args) ->
  rabbit_exchange:assert_args_equivalence(X, Args).

setup_schema() ->
  case mnesia:create_table(?RH_TABLE,
          [{attributes, record_info(fields, cached)},
           {record_name, cached},
           {type, set}]) of
      {atomic, ok} -> ok;
      {aborted, {already_exists, ?RH_TABLE}} -> ok
  end.

%%private
cache_msg(XName, Content) ->
  rabbit_misc:execute_mnesia_transaction(
    fun () ->
      Cached = get_msgs_from_cache(XName),
      store_msg(XName, Cached, Content)
    end).

get_msgs_from_cache(XName) ->
  rabbit_misc:execute_mnesia_transaction(
    fun () ->
      case mnesia:read(?RH_TABLE, XName) of
        [] ->
          [];
        [#cached{key = XName, content=Cached}] ->
          Cached
      end
    end).

store_msg(Key, Cached, Content) ->
  mnesia:write(?RH_TABLE,
    #cached{key     = Key,
            content = [Content|lists:sublist(Cached, ?KEEP_NB-1)]},
    write).

msgs_from_content(XName, Cached) ->
  lists:map(
```

```
    fun(Content) ->
        {Props, Payload} = rabbit_basic:from_content(Content),
        rabbit_basic:message(XName, <<"">>, Props, Payload)
    end, Cached).

deliver_messages(Queue, Msgs) ->
  lists:map(
    fun (Msg) ->
      Delivery = rabbit_basic:delivery(false, false, Msg, undefined),
      rabbit_amqqueue:deliver(Queue, Delivery)
    end,  lists:reverse(Msgs)).

queue_not_found_error(QName) ->
  rabbit_misc:protocol_error(
    internal_error,
    "could not find queue '~s'",
    [QName]).
```

现在你的交换器正在运行，该对它进行测试了。让我们编写发布者和消费者来试试看吧。

12.3.4 测试你的插件

为了测试自定义交换器，你将创建一对 PHP 脚本：其中一个是消费者，另一个是生产者。你的测试内容包括：开启消费者，然后在不同的终端窗口运行生产者，发布 100 条消息到最近历史交换器。期望的结果是消费者接收并消费了所有消息。在普通的 AMQP 场景下，你应该无法在服务器上再看到这些消息了。在我们的示例中，你使用了自定义交换器，在其缓存中应该仍有最近 20 条消息可用。为了证明这一点，你将在不同的窗口启动另一个消费者，并将队列绑定到交换器上。之后，你期望接收最近 20 条消息。让我们创建 `recent_history_consumer.php` 文件，并添加以下代码。

清单 12.12 Recent History Exchange 消费者

```
<?php

require_once('../path/to/lib/php-amqplib/amqp.inc');

define('HOST', 'localhost');
define('PORT', 5672);
define('USER', 'guest');
define('PASS', 'guest');
define('VHOST', '/');

$exchange = 'rh-exchange';
```

```
$conn = new AMQPConnection(HOST, PORT, USER, PASS, VHOST);
$ch = $conn->channel();

$ch->exchange_declare($exchange,
        'x-recent-history',
        false,
        true,
        false);

list($queue,,) = $ch->queue_declare('');

$ch->queue_bind($queue, $exchange);

$consumer = function($msg){
    echo $msg->body, "\t";
};

$ch->basic_consume(
        $queue,
        '',
        false,
        true,
        false,
        false,
        $consumer);

echo "consuming from queue: ", $queue, "\n";

function shutdown($conn, $ch){
  $ch->close();
  $conn->close();
}

register_shutdown_function('shutdown', $conn, $ch);

while(count($ch->callbacks)) {
    $ch->wait();
}
?>
```

❶ 声明最近历史交换器

❷ 将队列绑定到交换器上

❸ 订阅队列

❹ 等待发来的消息

这里的代码和之前你在本书中见到的消费者很相似。重要的细节之处在于当你声明交换器时❶，你通过指定类型为 `x-recent-history` 来告诉 RabbitMQ 你想要使用自定义交换器。记住如果你运行的服务器没有安装该插件的话，那么这段代码将会抛出异常并失败。这是因为 RabbitMQ 无法找到对应该交换器类型的模块。在创建交换器后，你声明了匿名队列并将其绑定到交换器上❷。然后你将 `$consume` 传递给 `basic_consume` 方法，开始从队列中消费消息❸。你的回调函数会将接收到的消息内容回显（echo）到 `STD_OUT` 上。最后，你在信道上等待发来的消息❹。

让我们继续编写生产者代码。创建 `recent_history_producer.php` 文件，并添加以下内容。

清单 12.13 Recent History Exchange 生产者

```
<?php

require_once('../path/to/lib/php-amqplib/amqp.inc');

define('HOST', 'localhost');
define('PORT', 5672);
define('USER', 'guest');
define('PASS', 'guest');
define('VHOST', '/');

$conn = new AMQPConnection(HOST, PORT, USER, PASS, VHOST);

$channel = $conn->channel();

for($i=0; $i<100; $i++) {
  $msg = new AMQPMessage('msg_'.$i,
        array('content_type' => 'text/plain'));
  $channel->basic_publish($msg, 'rh-exchange');
}

$channel->close();
$conn->close();
?>
```

❶ 发布100条消息

与消费者一样，这段代码和你在本书中创建的其他生产者很相似。在这里，你发送 100 条消息❶到之前脚本中创建的 `rh-exchange` 交换器。为了能够识别每一条消息，你使用循环变量来对消息进行标记。现在让我们开启三个终端窗口来测试这段代码。注意，由于之前调用了 `make run-in-broker`，此时 RabbitMQ 应该还运行着。

开启一个终端窗口，进入到保存之前 PHP 代码的文件夹，并输入

```
$ php recent_history_consumer.php
consuming from queue: amq.gen-C56UdaXBTQdIeSEkiUPiZQ==
```

这样就开启了一个消费者，并输出了在哪个队列上进行消费。注意，你机器上的队列名称可能会不太一样。

然后，开启另一个终端，启动生产者并向 RabbitMQ 发送 100 条消息。输入下列命令：

```
$ php recent_history_producer.php
```

如果一切正常，并且消息从交换器路由到了消费者的话，那么在第一个窗口你应见到如下输出：

```
$ php recent_history_consumer.php
consuming from queue: amq.gen-nipi9vrRRoEOr/ZKI1kuaw==
msg_0  msg_1  msg_2  msg_3  msg_4  msg_5  msg_6
msg_7  msg_8  msg_9  msg_10 msg_11 msg_12 msg_13
msg_14 msg_15 msg_16 msg_17 msg_18 msg_19
... omitted output
msg_71 msg_72 msg_73 msg_74 msg_75 msg_76 msg_77
msg_78 msg_79 msg_80 msg_81 msg_82 msg_83 msg_84
msg_85 msg_86 msg_87 msg_88 msg_89 msg_90 msg_91
msg_92 msg_93 msg_94 msg_95 msg_96 msg_97 msg_98
msg_99
```

现在，如果你切回最后的那个窗口，并启动第二个消费者，就应该可以获取最近20条消息。让我们试试吧：

```
$ php recent_history_consumer.php
consuming from queue: amq.gen-V9qcoRYbOuSnWmgNx7DfXg==
msg_80 msg_81  msg_82  msg_83  msg_84  msg_85  msg_86  msg_87
msg_88 msg_89  msg_90  msg_91  msg_92  msg_93  msg_94  msg_95
msg_96 msg_97  msg_98  msg_99  msg_80  msg_81  msg_82  msg_83
msg_84 msg_85  msg_86  msg_87  msg_88  msg_89  msg_90  msg_91
msg_92 msg_93  msg_94  msg_95  msg_96  msg_97  msg_98  msg_99
```

喔。试验成功了！让我们看看发生了什么。首先，你启动了消费者，它声明了rh-exchange交换器；然后消费者将一个匿名队列绑定到该交换器上，并订阅了该队列。之后你启动了单独的窗口，发送了100条消息，并如期望的那样，那些消息投递到了最初的那个消费者。最后，你在单独的窗口启动了另一个消费者，无须发送任何新的消息，最近20条消息就会投递给该消费者。让我们看一下图12.3，观察一下当同时运行的时候，这两个测试的输出。

你完成了创建自己插件的练习，也就是自定义交换器。现在该来停止用于测试插件的RabbitMQ实例了。为了做到这一点，在Erlang命令行上输入q().。注意，语句中最后的这个点“.”是必需的。如果没有包含该点的话，Erlang解释器就会继续等待输入。当回到shell命令行上的时候，你可以输入ls dist/来看看辛勤工作的产物：插件的.ez文件。

```
$ ls dist/
amqp_client-0.0.0.ez
rabbit_common-0.0.0.ez
rabbitmq_recent_history_exchange-0.1.0-rmq.ez
```

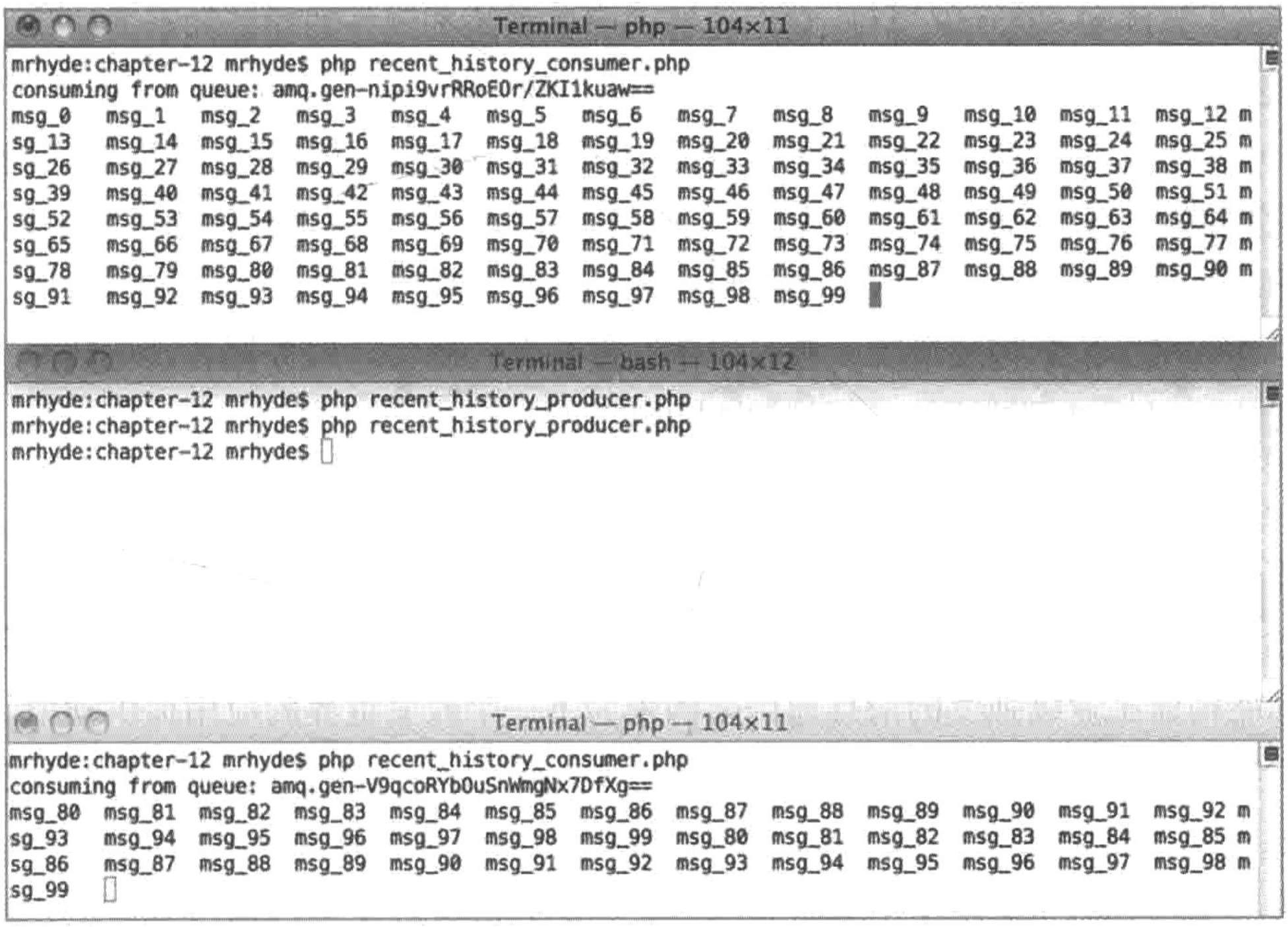

图 12.3 测试 Recent History Exchange

如果你想要将插件安装到服务器上的话，那么就将文件 rabbitmq_recent_history_exchange-0.0.0.ez 复制到 RabbitMQ 安装目录下的 plugins 文件夹下[7]，然后运行下列命令并重启服务器：

```
rabbitmq-plugins enable rabbitmq_recent_history_exchange
```

如果你希望移除插件的话，则首先要做的是删除用 x-recent-history 类型声明的所有交换器，然后你就能禁用插件并重启服务器了。如果需要的话，你可以从 plugins 文件夹中移除 rabbitmq_recent_history_exchange 插件文件。

12.4 总结

如果你认为会被 RabbitMQ 默认出厂设置限制住的话，那么本章就向你证明了事实并非如此。市面上有很多 RabbitMQ 的插件，包括官方支持的如管理

7 rabbit_common-0.0.0.ez和amqp_client-0.0.0.ez这两个文件仅在构建你的插件时才需要。由于这两个文件中的内容已经随RabbitMQ一起发布了，因此它们不需要部署到服务器的plugins文件夹下。

（Management）插件和 STOMP 插件，它们能够向服务器添加额外的功能和新的协议，以及社区提供的插件（如 Riak 交换器）。你也将 RabbitMQ 探了个究竟，并实现了自己的插件。当然，我们想带给你的不仅仅是 Hello World 之类的项目，所以你能创建自定义交换器。一路下来，你快速浏览了 Erlang 编程，之后如果你想要更深入地了解 RabbitMQ 源代码以学习它的内部行为的话，这对你很有帮助。为了创建插件，你学习了 RabbitMQ 公伞构建系统，它可以帮助你从源代码编译服务器和其他插件。总之，现在你可以按自己的想法来打造 RabbitMQ 了。

通过使用自定义代码来将 RabbitMQ 按照你的意愿进行打造的这种能力代表了 Rabbit 这座“大山”的巅峰，同时也为我们的旅程画上了圆满的句号。你从最简单的目的开始：将自己从紧耦合和应用间同步通信中解放出来。现在你能做的已远不止于此。你能构建工业级强度的消息通信架构来为下一个改变世界的应用提供动力。不管你是否在编写下一个成功的遛狗应用，改变医生帮助病人的方式，或是帮助旅途中的父亲在路上见见他的女儿，我们希望你能明白消息通信存在于软件的每个角落。最重要的是，我们希望这本书能够帮助你，以便你能专注于如何使用 RabbitMQ 而非自行深入研究。随着我们这次“旅途”的结束，你和 Rabbit 的旅途才刚刚开始，眼前是无尽的消息通信世界。如果这一路上你需要任何帮助的话，我们非常乐意在 *RabbitMQ in Action* 论坛上收到你的来信。尽情享受 RabbitMQ 之旅吧！

附录A
在Java和.NET上使用Rabbit

虽然我们选择 Python 和 PHP 作为教学语言是由于其语法清晰、适宜教学，但是我们也意识到了 .NET 和 Java 程序员的读者也很多。同样地，用于不同语言的 AMQP 客户端大多数在接口层面都十分相似。但是 Java 和 .NET 的客户端与其他语言迥然不同，以至于我们必须给出一些建议来指导本书中的示例映射到这两门语言上。记住这点，我们会将本书中的部分示例转换为 .NET 和 Java 平台上的等价物。特别是我们会向你展示用 C# 如何编写 Hello World 示例。同样地，我们会将第 4 章中的告警和 RPC 示例分别用 C# 和 Java 重写。在每个示例中，我们的目标是尽可能地遵循最初 Python 和 PHP 示例的结构、注释和命名约定。我们希望能够帮助你构建一副思维导图（mental map），当你在本书中寻找任何其他示例的时候，可以轻松将其转换成 Java 或者 .NET 平台对应的代码。像本书中的其他示例一样，所有附录的源代码可以在本书的 Github 代码库中进行下载：https://github.com/rabbitinaction/sourcecode。在该代码库中，你不仅可以找到 `.java` 和 `.cs` 源文件，也可以找到 .NET 示例的 Visual Studio 项目（可以随时用 `msbuild` 构建）。事不宜迟，让我们来看一下如何将第 2 章中的 Hello World 示例转换成 C# 吧。

A.1 用C#重写Hello World示例（库选项和Hello World程序）

在 .NET 世界中有太多的编程语言可供选择，我们将专注于 C# 下的 RabbitMQ.NET 客户端使用。特别是，本附录中的所有 .NET 示例都假设你使用 .NET 框架的 4.0 版本（包括 C# 4.0）。但是在我们开始动手编写 C# 版的 Hello World 之前，你首先要做的是安装 RabbitMQ.NET 客户端。

你总是可以在 http://www.rabbitmq.com/dotnet.html 上找到 .NET Rabbit 客户端的最新版本。在我们的示例中将使用 2.7.0 版本。在 .NET 客户端下载页面里，你可以找到自动安装包（.MSI）和 .ZIP 两种格式的客户端。为了方便起见，我们建议使用 .MSI，因此请下载 .MSI 安装包并运行（见图 A.1）。该示例假设客户端已经安装在了默认位置（C:\Program Files\RabbitMQ\DotNetClient）。

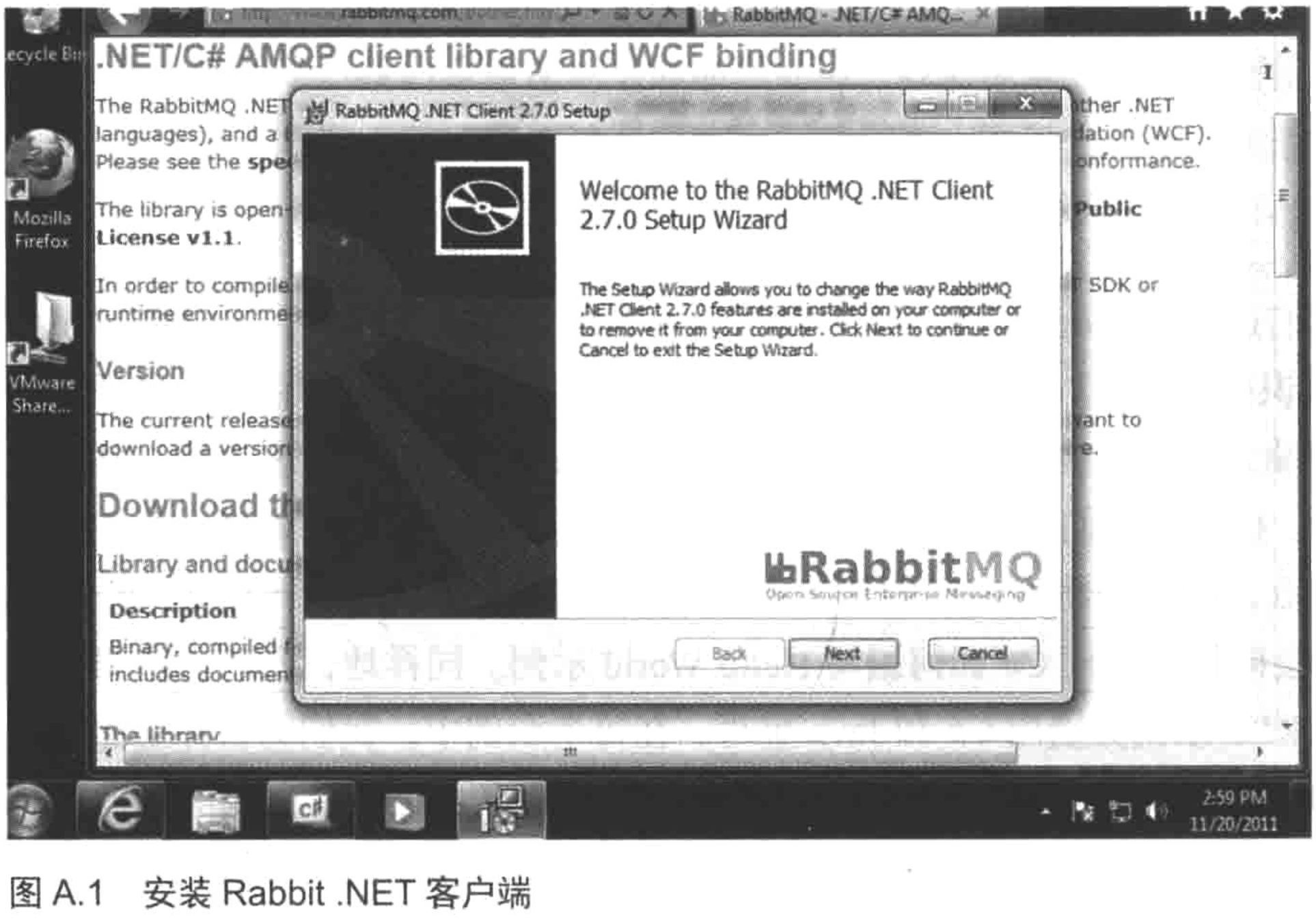

图 A.1 安装 Rabbit .NET 客户端

在安装好了客户端之后，为你的 Hello World 消费者创建 Visual Studio 项目（选择 Empty Project），如图 A.2 所示。

在开始编写代码之前，你需要做的最后一件事是将 RabbitMQ.NET 客户端的引用添加到项目中；否则，Visual Studio（或者 `msbuild`）将无法找到它。首先，在新项目下的 Solution Explorer 中右击 References 项，并选择 Add Reference（见图 A.3）。

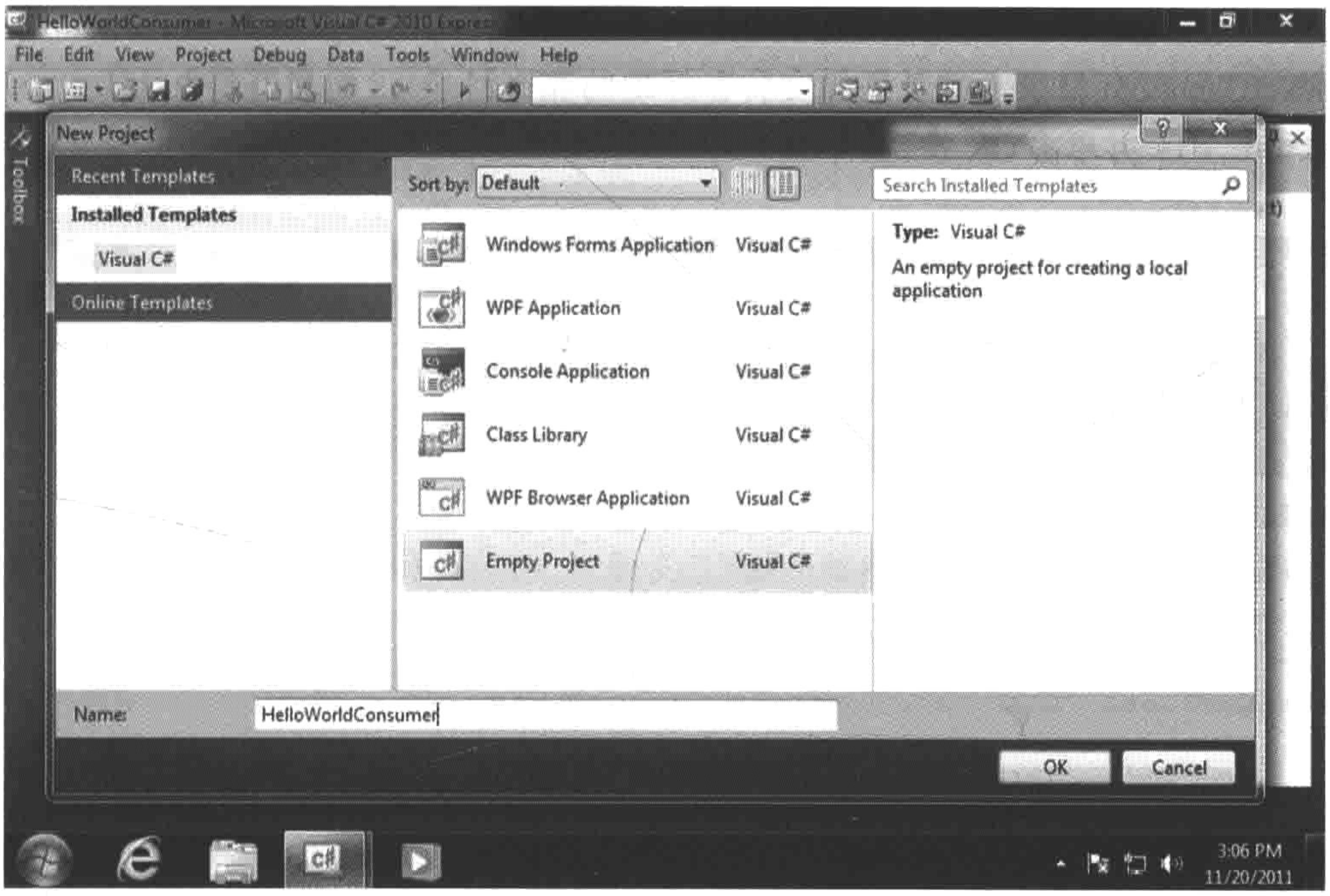

图 A.2 创建 HelloWorldConsumer 项目

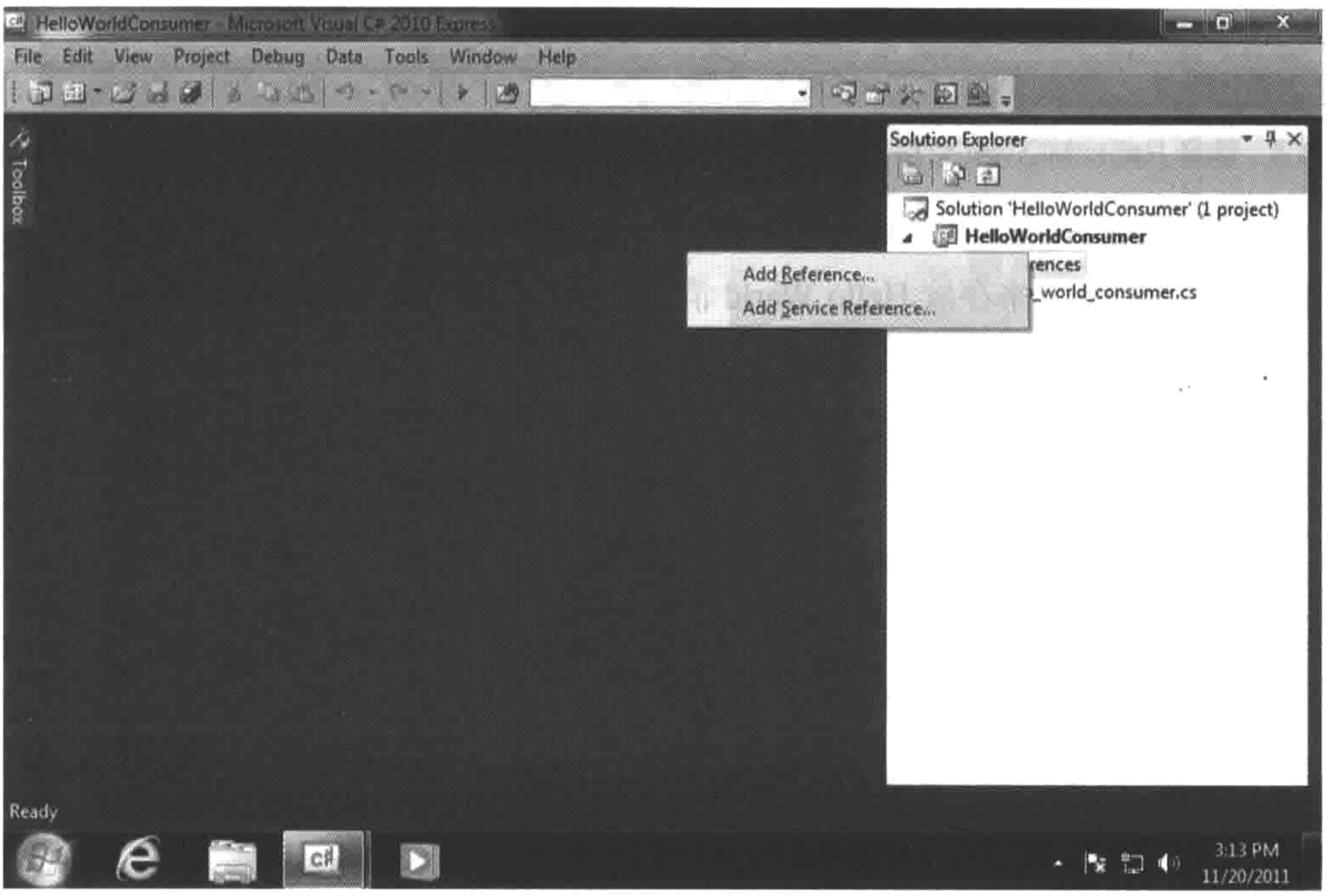

图 A.3 为 Visual Studio 项目添加引用

下一步选择 Browse 选项卡（见图 A.4），并导航到 C:\Program Files\RabbitMQ\DotNetClient\bin 目录（或者是你安装客户端的那个目录）。然后选择 `RabbitMQ.Client.dll` 并单击 OK 按钮。

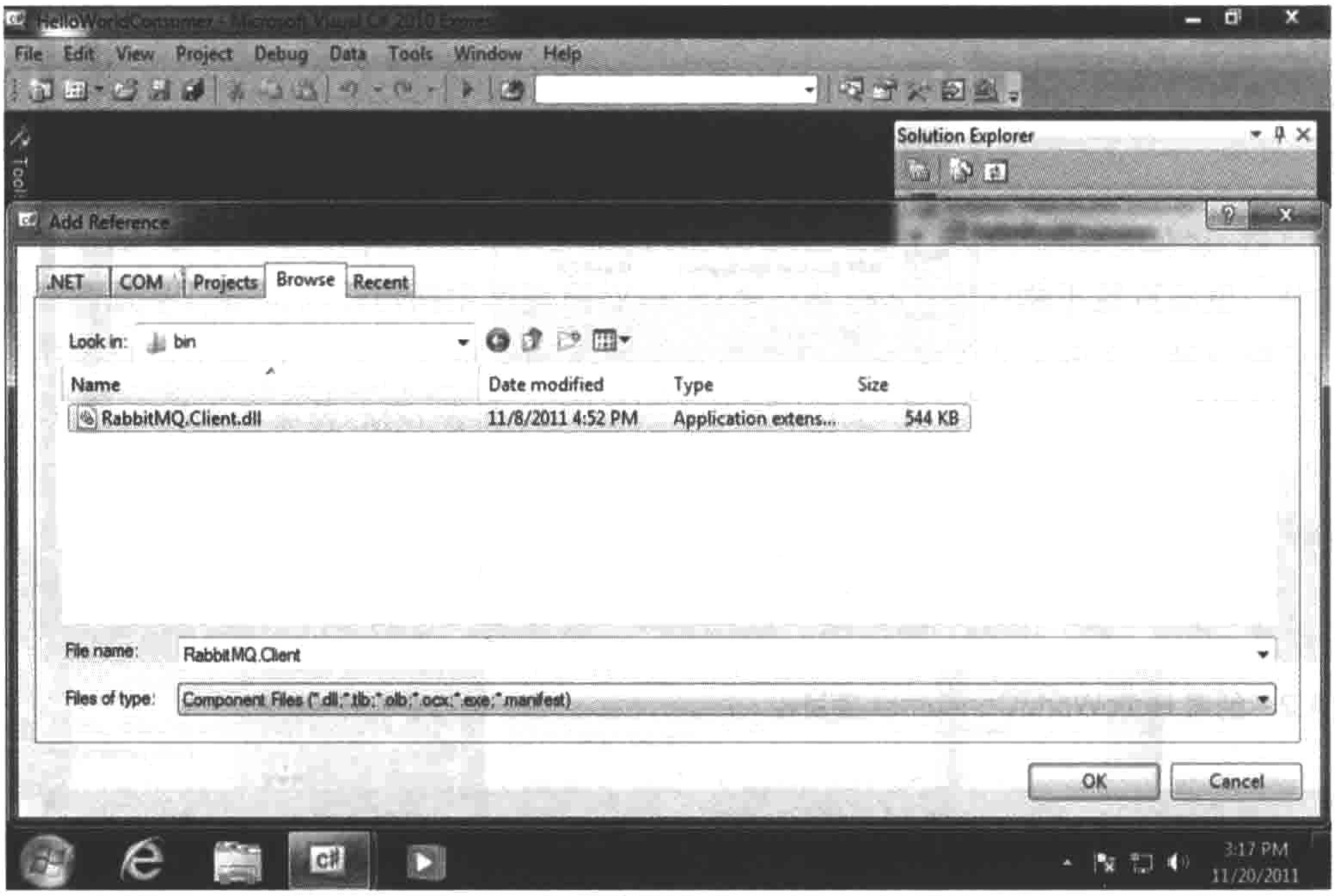

图 A.4 选择 RabbitMQ.Net 引用

在将 RabbitMQ.NET 添加到项目之后，在项目中新建一个文件并命名为 hello_world_consumer.cs，来存放 Hello World 消费者代码。首先在该文件中输入下列导入代码：

```
using System;
using System.Text;
using RabbitMQ.Client;
using RabbitMQ.Client.Events;
```

导入 RabbitMQ.Client 后，你就可以创建连接和信道了（IConnection 和 IModel），同时也能对消费订阅进行管理（QueueingBasicConsumer）。你同时需要导入 RabbitMQ.Client.Events，以便在消息投递事件发生时，能够对传回给消费者的参数进行访问。

由于 RabbitMQ 服务器和你的消费者很有可能不是运行在同一台机器上，因此你需要从命令行接收 Rabbit 主机地址参数：

```
namespace HelloWorld {
    class Consumer {

        public static void Main(string[] args) {

            if(args.Length < 1) {
                Console.WriteLine("Must supply hostname.");
                Environment.Exit(-1);
            }
```

现在，你已经开始着手编写 HelloWorld.Consumer 类来实现消费者了，并获得了 RabbitMQ 主机名用于连接。让我们连接到服务器并开始创建消息通信结构吧。

清单 A.1 为消费者创建消息通信结构

```
var conn_factory  = new ConnectionFactory();

conn_factory.HostName = args[0];
conn_factory.UserName = "guest";
conn_factory.Password = "guest";

IConnection
    conn = conn_factory.CreateConnection();
IModel chan = conn.CreateModel();

chan.ExchangeDeclare("hello-exchange",
                     ExchangeType.Direct,
                     true,
                     false,
                     null);

chan.QueueDeclare("hello-queue",
                  false,
                  false,
                  false,
                  null);

chan.QueueBind("hello-queue",
               "hello-exchange",
               "hola");
```

❶ 建立到服务器的连接

❷ 获取信道

❸ 声明交换器

❹ 声明队列

❺ 用键hola绑定队列和交换器

消费者首先做的是创建 ConnectionFactory 对象，用来产生并管理到服务器的连接。在使用工厂创建连接后❶，你使用返回的连接对象（conn）来创建信道❷。你可能注意到了 channel 对象（chan）是 IModel 类型，代表了 AMQP channel。这和任何其他 Rabbit 客户端库不一样（包括 Java 客户端），它们使用 channel 来代表

AMQP 信道[1]。所以当你看到 RabbitMQ.NET 客户端的类中使用 model 时，可别感到迷惑，在这里它（model）指代的就是信道。

在创建了 channel 之后，你准备为 Hello World 消费者和生产者来声明交换器和队列以构建通信结构。如同第 2 章中的原始 Hello World 消费者那样，你定义了 direct 类型的交换器，设置成可持久化但非自动删除。`ExchangeDeclare` 函数的第二个参数指明了交换器类型为 direct（`ExchangeType.Direct`）❸。在 RabbitMQ.NET 客户端中，交换器类型是使用 `ExchangeType` 类的常量来指明的。如果你想要将 `hello-exchange` 声明为 fanout 的话，则可以使用 `ExchangeType.Fanout`[2]。

然后，你使用 `QueueDeclare` 来创建❹非持久化的、非自动删除的队列（`hello-queue`），并在路由键 `hola` 上绑定到 `hello-exchange` ❺。

最终，你准备订阅 `hello-queue` 并开始处理消息。

清单 A.2 订阅和处理消息

```
QueueingBasicConsumer
    consumer = new QueueingBasicConsumer(chan);
String consumer_tag = chan.BasicConsume("hello-queue",
                                        false,
                                        consumer);

while(true) {
    BasicDeliverEventArgs
        evt_args = (BasicDeliverEventArgs)
                        consumer.Queue.Dequeue();
    IBasicProperties msg_props = evt_args.BasicProperties;
```

❶ 订阅消费者

❷ 开始消费

❸ 处理发来的消息

---

1 在.NET客户端上使用model来描述信道是AMQP早期的历史遗留问题。当时有这样的想法，除了TCP协议之外的其他传输协议（HTTP、SCTP等）可能会用于AMQP。由于channel的概念是特定于TCP传输层的，因此.NET客户端架构使用了更为通用的术语model来代表相同的概念。现在TCP是AMQP的唯一传输协议，所有新的客户端都使用术语“信道”（channel）。

2 `ExchangeType`是一个方便类。你也可以选择性地使用包含交换器类型的字串来传入`ExchangeDeclare`。举个例子，你可以用`"direct"`替换`ExchangeType.Direct`来传入`ExchangeDeclare`。

```
String
    msg_body = Encoding.ASCII.GetString(evt_args.Body);

chan.BasicAck(evt_args.DeliveryTag,
                  false);

     if(msg_body == "quit") {
         chan.BasicCancel(consumer_tag);
         break;
     } else
                  Console.WriteLine("Message Body: " + msg_body);

         }

         Environment.Exit(0);
         }
    }
}
```

❹ 消息确认

❺ 停止消费消息并退出

在消费者的最后一段还有些问题没有敲定，所以让我们分别进行讨论。首先你使用 QueueingBasicConsumer 方便类来设置消费者对象（consumer）❶。QueueingBasicConsumer 分离了订阅消息的到达和消息的处理。当信道对象接收到 consumer 订阅的新消息时，它会触发 consumer 的 HandleBasicDeliver 方法。该方法会接收消息并将其装入 consumer 对象内部的一个 SharedQueue 实例。这意味着，新的消息可以源源不断地流向 consumer，而不会被实际处理 SharedQueue 中的消息代码阻塞。在创建完 consumer 之后，你将 consumer 作为参数，通过调用信道的 BasicConsume 方法开始消费消息。这告诉信道去订阅 hello-queue，并在订阅的消息到达时，将它们装入 consumer 的 SharedQueue 中。

虽然 consumer 处理接收订阅消息这类繁重的工作，但是你仍然需要自行处理消息，并向服务器发回确认消息。在 Python 的 Pika 客户端中，你创建一个函数来进行处理，并将其作为回调函数传给 basic_consume。但是如你所见❶，QueueingBasicConsumer 不使用回调函数 / 基于事件的模型。取而代之的是，当消息到达时，它们会被存放到 consumer 的 SharedQueue 中，并交由你来对队列进行轮询，获取消息并进行处理。在该示例中，你的轮询机制是一个❷无限 while 循环（while(true){}），它将不停地从 consumer 获取下一条消息，对其进行处理，然后重新开始。为了从队列中移除消息❸，你可以调用 consumer 的 Queue 属性的 Dequeue() 方法。它会返回一个 BasicDeliverEventArgs 对象（evt_args），包含了 AMQP 消息属性（包括消息头）、消息体，以及用来确认消息

用的投递标记（delevery tag）。得到 evt_args 后，你将其分解成两块：一块是 IBasicProperties 对象（msg_props），包含了消息的头和属性（例如 .ContentType 和 .DeliveryMode）；另一块是消息体（msg_body），在本例中是一个 ASCII 编码的字串。

BasicConsume

你可以多次使用不同的队列作为参数调用 BasicConsume，来订阅同一个消费者对象。这会导致来源于不同订阅的所有消息都将放置到消费者对象的同一个 SharedQueue 上。然后在处理过程中，你就可以使用每条消息携带的 BasicDeliverEventArgs 对象来确定消息是来源于哪个队列的（因为它们在 SharedQueue 中混在了一起）。你更愿意使用上述方法，因为 consumer.Queue.Dequeue() 在 SharedQueue 为空时，会停止执行并等待新的消息。因此，如果你有多个对不同消费者对象的 Dequeue() 方法进行连续调用的话，那么对第一个消费者对象的 Dequeue() 会阻塞之后的消费者对象 Dequeue() 方法的调用，直到第一个消费者对象 / 订阅接收到消息为止。通过为多个订阅使用单一的消费者对象，你只需要一个 Dequeue() 调用来服务所有订阅，从而避免了相互阻塞。

到现在为止，你已经成功地解码了消息体，并提取了它的属性，因此你可以通过调用 chan.BasicAck 方法，以消息的投递标记（evt_args.DeliveryTag）作为参数，告诉 Rabbit 该消息成功消费了❹。你可能注意到了，传递给 BasicAck 的第二个参数为 false。这告诉 BasicAck 每次确认一条消息。最后，还要做的是将解码消息的内容打印给用户；或者如果消息包含 quit 关键字的话，就让程序退出❺。

在将 Hello World 消费者转换为 .NET 后，你需要将生产者也转成 .NET，这样你才有消息消费！就和所有这些例子一样，生产者的代码要比消费者的简单多了。你将和之前那样向 Producer 类中导入相同的包（你可以在新的 Visual Studio 项目中创建 hello_world_producer.cs 文件，或者在消费者所在的那个项目下创建也可以）。

清单 A.3 在 .NET 平台上发布消息

```
namespace HelloWorld {
    class Producer {

        public static void Main(string[] args) {

            if(args.Length < 2) {
               Console.WriteLine("Must supply hostname and " +
                                 "message text.");
               Environment.Exit(-1);
            }

            var conn_factory  = new ConnectionFactory();

            conn_factory.HostName = args[0];
            conn_factory.UserName = "guest";
            conn_factory.Password = "guest";

            IConnection conn = conn_factory.CreateConnection();
            IModel chan = conn.CreateModel();

            chan.ExchangeDeclare("hello-exchange",
                                 ExchangeType.Direct,
                                 true,
                                 false,
                                 null);

            string msg_body = args[1];
            IBasicProperties msg_props = chan.CreateBasicProperties();
            msg_props.ContentType = "text/plain";

            chan.BasicPublish("hello-exchange",
                              "hola",
                              msg_props,
                              Encoding.ASCII.GetBytes(msg_body));

            Environment.Exit(0);
        }
    }
}
```

❶ 接收服务器地址和消息内容

❷ 建立到服务器的连接

❸ 声明交换器

❹ 创建纯文本消息

❺ 发布消息

除了用于连接的服务器地址外，你也需要从命令行获取用于发布的消息内容❶。之后，你准备好连接服务器❷，并声明了 hello-exchange ❸用来发布消息。创建消息❹并发布的步骤和用来处理消息的步骤很相似，只不过将顺序反了一下。首先，你从命令行获取消息内容，并存储到字串 msg_body 上。下一步，你创建一个 IBasicProperties 对象（msg_props）来存储消息的发布属性，并设置消息的内容类型为 text/plain。这样做使得消费者可以知道消息体是纯 ASCII 文本。最后，你将将路由键设置为 hola ❺，把消息内容（msg_body）和设置在 msg_props 的消息属性一并发往 hello-exchange。你也许注意到了作为消息发布的一部分，你首先使用 Encoding.ASCII.GetBytes 将消息转换为了一个字节数组。AMQP（以

及作为代表的 RabbitMQ）对于消息体的内容一无所知。唯一的要求是内容必须是 8 位字节序列。因此，Rabbit.NET 客户端的 `BasicPublish` 命令只接收字节数组作为消息体。

在完成了C#版的消费者和生产者之后，让我们构建并进行测试吧！在该示例中，RabbitMQ 位于一个不同的虚拟机上（`192.168.241.1`），而不是你的消费者和生产者所在的 Windows 系统上。首先打开两个 PowerShell 实例[3]（见图 A.5），并将其中之一导航到 Hello World 消费者项目上，同时另一个导航到 Hello World 生产者上。首先进行构建，并启动消费者。

```
PS > msbuild
Microsoft (R) Build Engine Version 4.0.30319.1
[Microsoft .NET Framework, Version 4.0.30319.1]
Copyright (C) Microsoft Corporation 2007. All rights reserved.
...
Done Building Project "\HelloWorldConsumer.csproj" ...

Done Building Project "\HelloWorldConsumer.sln" (default targets).

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:00.17
PS > .\bin\Debug\HelloWorldConsumer.exe 192.168.241.1
```

现在，在另一个终端上构建生产者并发送一段测试消息：

```
PS \HelloWorldProducer> msbuild
Microsoft (R) Build Engine Version 4.0.30319.1
[Microsoft .NET Framework, Version 4.0.30319.1]
Copyright (C) Microsoft Corporation 2007. All rights reserved.
...
Done Building Project "\HelloWorldProducer.csproj" ...

Done Building Project "\HelloWorldProducer.sln
" (default targets).

Build succeeded.
    0 Warning(s)
    0 Error(s)
```

3 你可能需要将.NET framework完整路径添加到系统环境变量PATH中。举例来说，用于该示例的测试系统上的.NET framework路径为C:\Windows\Microsoft.NET\Framework\v4.0.30319。

```
PS > .\bin\Debug\HelloWorldProducer.exe 192.168.241.1 \
     "Hello there world!"
PS >
```

图 A.5 打开终端，用于测试生产者和消费者

回到消费者终端，消息到达了吗？

```
PS > .\bin\Debug\HelloWorldConsumer.exe 192.168.241.1
Message Body: Hello there world!
```

嗯，不错！ C# 版生产者和消费者可以工作了！由于 C# 版 Hello World 应用是由原始 Python 应用直接演变而来的，因此它们可以互相合作！为了尝试一些跨语言通信的好处（见图 A.6），试试用第 2 章的 `hello_world_producer.py` 向正在运行的 C# 版消费者发布一条消息吧！

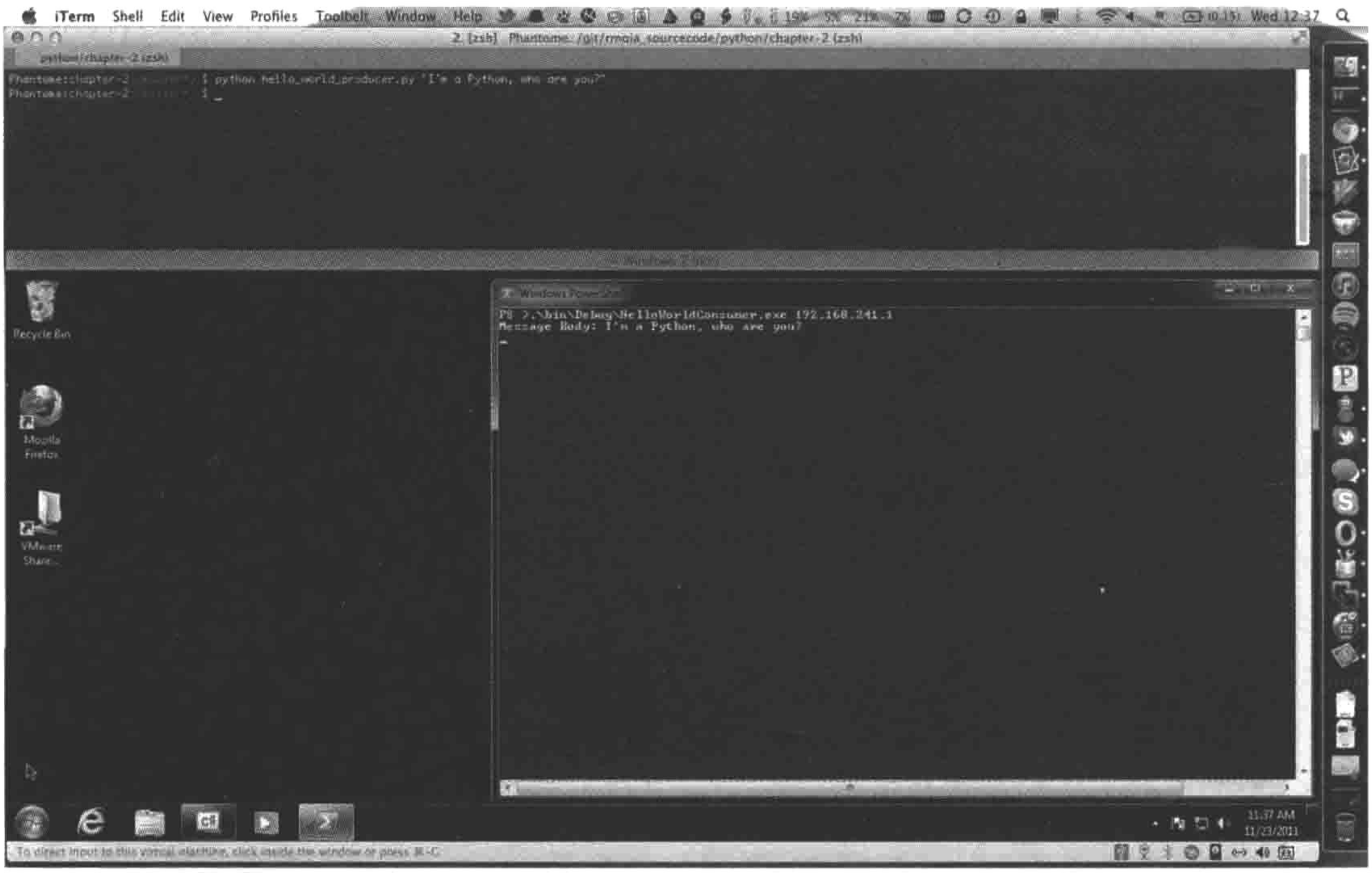

图 A.6 跨平台的消息通信实战！

现在你掌握了在 .NET 平台上使用 RabbitMQ 的基础，让我们快速地把在第 4 章编写的告警服务器转换成 C# 版本吧。

A.2 再访告警系统：将告警应用迁移到面向事件的.NET平台

在学习了 C# 版的 Hello World 之后，你也许会想我们已经为你讲解了足够多的 RabbitMQ.NET 基础，你能在脑子里对本书剩余的例子进行转换了。你答对了 90%。我们没有提到的是如何使用基于事件的方法来消费消息。在 Python 的客户端 Pika 上，所有的消费者均用回调函数（面向事件的）为每一个订阅进行注册，并在之后当订阅的消息到达时，函数被触发。就像你在 C# Hello World 消费者上看到的那样，它使用 `QueueingBasicConsumer` 来管理订阅和消息消费，因此需要的是一个基于轮询的方法。但是如果你是一位 .NET 开发人员，并且像我们一样更偏好面向事件

编程的话，那你就走运了——通过使用 EventingBasicConsumer[4]，Rabbit.NET 支持面向事件的消费模型。为了演示如何在 .NET 中使用面向事件的消费接口，我们将第 4 章的告警服务器示例转换为 C# 版本，并展示相关部分。如果要看告警服务器消费者（以及生产者）的完整 C# 版本的话，请从伴随本书的示例代码库中检出这部分代码：https://github.com/rabbitinaction/sourcecode。

在我们开始之前，由于原始的告警示例使用 JSON 通信，因此你需要选择一个 .NET JSON 库。在该示例中，你将使用由 James Newton-King 编写的 JSON.NET 4.0 库：http://json.codeplex.com/。在从 Codeplex 安装了最新版本之后，请将 JSON.NET 库的引用添加到你的项目中（默认的安装路径位于 C:\Program Files\JSON.NET\Bin\Net35）。

在 .NET 中，转变为面向事件的消费模型的最主要改变在于使用 EventingBasicConsumer 作为消费者对象，而非 QueueingBasicConsumer：

```
EventingBasicConsumer
    c_consumer = new EventingBasicConsumer {Model = chan};
```

注意，由于 EventingBasicConsumer 没有定义构造函数，因此你在实例化的时候将 c_consumer 的 Model 属性直接设置引用为信道对象（chan）。为什么说 EventingBasicConsumer 是面向事件的原因在于它在内部使用了 C# 的事件和委托，并在接收到消费者订阅的消息时，触发一个预定义的回调函数列表。为了将函数（在本例中为 critical_notify）添加到回调函数列表，你需要将它添加到消费者对象的 Received 属性，然后调用 BasicConsume：

```
c_consumer.Received += critical_notify;
chan.BasicConsume("critical",
                  false,
                   c_consumer);
```

对 EventingBasicConsumer 回调函数的唯一需求是它的返回值为 void，并且接收 IBasicConsumer 和 BasicDeliverEventArgs 对象作为参数。让我们看看 C# 版本的 critical_notify 长什么样子。

4 不幸的是，RabbitMQ的Java客户端没有可用的面向事件的消费接口。

清单 A.4 Critical 通知订阅回调处理函数

```
private static void critical_notify(IBasicConsumer consumer,
                                    BasicDeliverEventArgs eargs) {

    string[] EMAIL_RECIPS = new string[]
                            {"ops.team@ourcompany.com"};

    IBasicProperties msg_props = eargs.BasicProperties;
    String msg_body = Encoding.ASCII.GetString(eargs.Body);

    msg_body = JsonConvert.DeserializeObject
                          <string>(msg_body);

    send_mail(EMAIL_RECIPS,
              "CRITICAL ALERT",
              msg_body);

    Console.WriteLine("Sent alert via e-mail! Alert Text: " +
                      msg_body + " Recipients: " +
                      string.Join(",", EMAIL_RECIPS));

    consumer.Model.BasicAck(eargs.DeliveryTag,
                            false);
}
```

❶ 获取消息体和属性

❷ 使用JSON解码消息体

❸ 将解码的消息内容用邮件发送

❹ 确认消息

对于在 critical 队列上收到的每一条消息（如第 4 章原始版本一样），EventingBasicConsumer 都会触发 critical_notify 回调函数。EventingBasicConsumer 对象通过 consumer 参数将自身的引用传入，同时通过 eargs 参数将收到的消息拷贝传入。像之前那样，你将从中抽取出❶消息属性和消息体。但是由于消息体实际上是 JSON 编码的，因此你需要使用 JSON.NET 库❷来将消息体解码为 .NET 数据类型。之后你将发送告警邮件❸并通知相关负责人。有意思的地方在于❹你如何确认消息。由于 BasicAck 是信道对象的方法，因此你需要访问该对象以确认消息。你通过访问传入的 consumer 对象上的 Model 属性来做到这一点。你可能记得，.NET 客户端将 AMQP 信道称为模型（model），同时 consumer 的 Model 属性包含了指向接收消息的信道的引用。因此，对消息的确认就等同于简单地调用 consumer.Model.BasickAck()，同时提供包含在 eargs 内的消息的 DeliveryTag。这就是为 EventingBasicConsumer 编写回调函数的全部内容。

在 .NET 中使用面向事件的消息消费模型真的如此简单。在本书的示例代码库中可以检出该 C# 版本消费者的完整源代码。现在，.NET 介绍得差不多了；让我们也给 Java 多一些关爱，来看看在这个世界上最流行的字节码解释器上如何使用 Rabbit。

A.3 边喝咖啡边RPC：用Java实现AMQP RPC

在本节中，你将在 Java 平台上使用 RabbitMQ 客户端来实现 4.3 节的 RPC 客户端和服务器。客户端会发送一条以 JSON 格式编码的消息，内容包含了客户端版本和当前时间戳。消息看起来像这样：

```
{"client_name": "RPC Client 1.0",
"time" : 1320846509}
```

当请求发送给服务器后，客户端就会等待服务器的响应。服务器会将 JSON 对象从 AMQP 消息中取出，抽取消息的时间戳，然后它会用单词 Pong！和客户端发来的原始时间戳作为回应[5]。

A.3.1 获取 Java 库

首先，你要做的工作是下载最新版本 RabbitMQ 的 Java 客户端。你可以直接访问客户端下载页，然后选择适用于你的平台的包：http://www.rabbitmq.com/java-client.html。

对该示例来说，你将下载的包名为：Binary, compiled for Java 1.5 or newer（zip）。创建一个名为 `java-rpc` 的文件夹，然后使用 `wget` 来下载这个库。或者你也可以单击连接，使用浏览器将文件下载到该文件夹下。

```
$ mkdir java-rpc
$ cd java-rpc
$ wget http://www.rabbitmq.com/releases/rabbitmq-java-client/\
v2.7.0/rabbitmq-java-client-bin-2.7.0.zip
```

现在下载好库之后，你将它解压，然后将 `*.jar` 复制到应用程序的 `lib` 目录下。

```
$ unzip rabbitmq-java-client-bin-2.7.0.zip
$ mkdir lib
$ cp rabbitmq-java-client-bin-2.7.0/*.jar lib/
```

---

5 在本节中，你将实现RPC客户端和服务器来说明如何将第4章的示例进行转换。话虽如此，值得一提的是官方RabbitMQ Java客户端实现了基本RPC功能，请参考http://www.rabbitmq.com/api-guide.html#rpc。

由于你也需要发送 JSON 格式的消息，因此下载 Douglas Crockford 的 org.json Java 库：

```
$ wget http://search.maven.org/remotecontent\
?filepath=org/json/json/20090211/json-20090211.jar \
-O lib/json-20090211.jar
```

输入 ls lib 命令，你应该看到下列文件：

```
$ ls lib/
commons-cli-1.1.jar        commons-io-1.2.jar
json-20090211.jar          junit.jar
rabbitmq-client-tests.jar rabbitmq-client.jar
```

A.3.2 设置 class path

你应该知道，每次运行 Java 程序，你都需要指明 class path，因而 JVM 才能知道到哪里找程序需要的包和类。class path 很容易就会变得越来越长，因此你应该创建一个 shell 变量来存放 class path 信息，这样你就不用一次次地重复输入了。在类 UNIX 系统上，你可以像这样来创建变量：

```
$ export CP=.:./lib/commons-io-1.2.jar:./lib/commons-cli-1.1.jar:\
./lib/rabbitmq-client.jar:./lib/json-20090211.jar
```

在 Windows 系统上，你需要将冒号替换为分号来分隔这些 .jar 文件，并用 set 命令来替换 export 命令：

```
set CP=.;./lib/commons-io-1.2.jar;./lib/commons-cli-1.1.jar;\
./lib/rabbitmq-client.jar;./lib/json-20090211.jar
```

每次运行 Java 版消费者或者生产者时，都将使用该 class path 变量。在设置完基础信息之后，该来编写 RPC 服务器端和客户端了。在本节的剩余部分，你会直接接触 Java 代码。如果你想要了解更多库的内容的话，则可以阅读在线 API 指引：http://www.rabbitmq.com/api-guide.html，同时 Javadocs 位于 http://www.rabbitmq.com/releases/rabbitmq-java-client/v2.7.0/rabbitmq-java-client-javadoc-2.7.0/。

A.3.3 创建 RPC 服务器

和所有 Java 程序一样，首先你需要导入程序用到的类，因此在 java-rpc 文件夹下创建名为 Client.java 的文件。我们会一步步地讲解该类的代码，并在最终向你展示整个源文件。让我们从讲解 import 指令开始：

```
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.QueueingConsumer;
import com.rabbitmq.client.QueueingConsumer.Delivery;
import com.rabbitmq.client.AMQP.BasicProperties;
import org.json.JSONObject;
```

在这段代码片段中，你导入了RabbitMQ客户端的ConnectionFactory、Connection和Channel类。这些类用于建立到服务器的连接，并获取信道。类QueueingConsumer和QueueingConsumer.Delivery分别用于获得RabbitMQ消费者和管理消息投递。AMQP.BasicProperties类用于创建消息的AMQP属性。最后包含进来的来自org.json包的JSONObject类是用来在内存中加载JSON对象的。

现在让我们看看类中的init()方法，在该方法中创建了AMQP连接，获取了一条信道，然后使用该信道声明交换器、队列，并最终将它们绑定到一起，以此构建AMQP通信结构。以下是代码清单。

清单A.5 服务器端init方法

```
public Server init()
throws Exception {
  ConnectionFactory factory = new ConnectionFactory();
  factory.setUsername("rpc_user");
  factory.setPassword("rpcme");
  connection = factory.newConnection();

  channel = connection.createChannel();

  channel.exchangeDeclare("rpc", "direct");
  channel.queueDeclare("ping", false, false, false, null);
  channel.queueBind("ping", "rpc", "ping");

  consumer = new QueueingConsumer(channel);
  channel.basicConsume("ping", false, "ping", consumer);

  System.out.println(
    "Waiting for RPC calls..."
  );

  return this;
}
```

❶ 创建连接

❷ 创建信道

❸ 设置AMQP通信结构

❹ 开始消费

首先你要做的是创建ConnectionFactory类的实例❶，用来建立连接。就像你看到的那样，factory接受像setUsername和setPassword这样的方法调

用。你可以通过这些方法来提供所需的连接信息。ConnectionFactory 类也有像 setVirtualHost 之类的方法。在这里你使用 rpc_user 作为用户名、rpcme 作为密码进行连接。一旦 factory 设置完毕，你就能调用 newConnection 方法来获取连接对象了，获得连接之后可以得到 channel 对象❷。就像你看到的那样，你没有声明 connection 或者 channel 变量，因为你已经把它们添加到类的成员变量中了。

然后你使用信道来设置 AMQP 通信结构❸。首先通过调用信道对象的 exchangeDeclare 方法来声明交换器。传递给该方法的参数为交换器名称和类型。然后你声明一个名为 ping 的队列。剩余的几个参数分别代表 durable、exclusive、autodelete 和 extra。在这里也就是说，你创建了一个非持久化、非自动删除、非排他性的队列。在队列创建之后，通过使用 ping 作为路由键将队列绑定到 ping 交换器上。

init 方法的最后一步是启动消费者。首先，通过传入 channel 对象获取了 QueueingConsumer 的一个实例❹。然后，通过调用 basicConsume 方法订阅到 ping 队列。在这里你使用字串 ping 作为消费者标记，并将消费者对象作为消息回调函数传入，因此每当有新的消息投递过来的时候，就会传给该消费者对象。basicConsume 方法神秘的第二个参数指明了消息的消费处于非自动确认模式——你需要为收到的每一条消息发送一个消息确认。

清单 A.6 服务 RPC 请求

```
public void serveRequests() {
  while (true) {
    try {

      Delivery delivery = consumer.nextDelivery();
      BasicProperties props = delivery.getProperties();

      channel.basicAck(delivery.getEnvelope().getDeliveryTag(),
                       false);
      System.out.println(
        "Received API call...replying..."
      );

      channel.basicPublish(
        "",
        props.getReplyTo(),
```

❶ 获得文本消息

❷ 确认消息

❸ 向客户端回应

```
        null,
        getResponse(delivery).getBytes("UTF-8")
      );

      } catch (Exception e){
        System.out.println(e.toString());
      }
    }
}
```

在该方法中你输入了一个无限循环，每次处理一条消息。通过调用消费者对象的nextDelivery方法来获得服务器发送的最新一条消息❶。消息投递对象同时拥有消息载荷和消息属性，在之后的方法中会用到。然后你通过调用basicAck❷，传入消息的投递标记，向服务器确认了该消息。其中投递标记是通过delivery获取消息信封，然后通过链式调用getDeliveryTag获得的。你也可以通过调用Envelope对象的getExchange获得用于路由消息的交换器，还可以通过调用getRoutingKey来获得消息路由键，等等。

最后，你通过调用basicPublish方法向客户端发送一条消息❸作为回应。消息将发往匿名交换器，并以原始客户端请求中的reply_to属性作为路由键。null参数表示该消息不是强制的。响应消息是通过调用方法getResponse创建的。该getResponse方法接下来就会实现。总结一下整个方法的内容：首先获得消费者中的下一条delivery。使用该delivery来获得消息属性。通过使用消息信封获得投递标记，然后对消息进行确认，之后你向客户端发送一条回应。现在让我们看看getResponse方法的代码。

清单A.7 创建RPC响应

```
private String getResponse(Delivery delivery) {
  String response = null;
  try {
    String message = new String(delivery.getBody(), "UTF-8");

    JSONObject jsonobject = new JSONObject(message);

    response = "Pong!" + jsonobject.getString("time");
  }
  catch (Exception e){
    System.out.println(e.toString());
    response = "";
  }
  return response;
}
```

❶ 获取消息体并转换为String类型

❷ 将消息转换为JSON对象

❸ 获得时间

该方法接收 Delivery 对象作为参数，因此它可以从中获得消息体❶，并将其作为 String 类型存放在 message 变量中。下一步要做的是将该字串解析为一个 JSON 对象，这样你就能获取由客户端发来的时间戳了❷，然后将该时间戳和字串 Pong! 一同发送回去❸。通过调用 JSONObject 实例的 getString 方法来获取其参数，它接收对象属性关键字作为参数并返回对应的值。

最后，让我们看看 Server 类的 main 方法，在该方法中你实例化了服务器来等待客户端的请求。以下是代码清单。

清单 A.8 服务器端 main 方法

```
public static void main(String[] args) {
  Server server = null;
  try {
    server = new Server();                                  <-- 服务器初始化
    server.init().serveRequests();
  } catch(Exception e) {
    e.printStackTrace();
  } finally {
    if(server != null) {
      server.closeConnection();                             <-- 资源清理
    }
  }
}
```

代码很简单。除了 try/catch/finally 逻辑之外，你只是创建了 Server 类的实例，通过调用 init 方法进行初始化，并在最后链式调用方法 serveRequests 来启动消息处理。类的完整代码如下所示，包含了在 finally 块中调用 closeConnection 方法。

清单 A.9 RPC 服务器完整代码

```
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.QueueingConsumer;
import com.rabbitmq.client.QueueingConsumer.Delivery;
import com.rabbitmq.client.AMQP.BasicProperties;
import org.json.JSONObject;

public class Server
{
  private Connection connection;
  private Channel channel;
  private QueueingConsumer consumer;
```

```
public Server Server(){
  return this;
}

public Server init()
throws Exception {
  ConnectionFactory factory = new ConnectionFactory();
  factory.setUsername("rpc_user");
  factory.setPassword("rpcme");
  connection = factory.newConnection();
  channel = connection.createChannel();
  channel.exchangeDeclare("rpc", "direct");
  channel.queueDeclare("ping", false, false, false, null);
  channel.queueBind("ping", "rpc", "ping");

  consumer = new QueueingConsumer(channel);
  channel.basicConsume("ping", false, "ping", consumer);

  System.out.println(
    "Waiting for RPC calls..."
  );

  return this;
}

public void closeConnection() {
  if (connection != null) {
    try {
      connection.close();
    }
    catch (Exception ignore) {}
  }
}

public void serveRequests() {
  while (true) {
    try {

      Delivery delivery = consumer.nextDelivery();
      BasicProperties props = delivery.getProperties();

      channel.basicAck(delivery.getEnvelope().getDeliveryTag(),
                       false);
      System.out.println(
        "Received API call...replying..."
      );

    channel.basicPublish(
      "",
      props.getReplyTo(),
      null,
      getResponse(delivery).getBytes("UTF-8")
    );

} catch (Exception e){
System.out.println(e.toString());
}
```

```
    }
  }

  private String getResponse(Delivery delivery) {
    String response = null;
    try {
      String message = new String(delivery.getBody(), "UTF-8");
      JSONObject jsonobject = new JSONObject(message);
      response = "Pong!" + jsonobject.getString("time");
    }
  catch (Exception e){
    System.out.println(e.toString());
      response = "";
    }
    return response;
  }

  public static void main(String[] args) {
    Server server = null;
    try {
      server = new Server();
      server.init().serveRequests();
    } catch(Exception e) {
      e.printStackTrace();
    } finally {
      if(server != null) {
        server.closeConnection();
      }
    }
  }
}
```

现在你已经完全实现了服务器端，让我们通过下列命令来编译它：

```
$ javac -cp ./lib/rabbitmq-client.jar:./lib/json-20090211.jar \
  Server.java
```

该命令会创建 Server.class 文件：

```
$ ls Server.class
Server.class
```

A.3.4 创建 RPC 客户端

让我们继续开始编码客户端吧。创建一个名为 Client.java 的文件，并往里面添加代码。像往常一样，完整的源代码会在本节的最后部分给出。首先需要加入的代码是下列 import 列表：

```
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.Connection;
```

```
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.QueueingConsumer;
import com.rabbitmq.client.QueueingConsumer.Delivery;
import com.rabbitmq.client.AMQP.BasicProperties;
import org.json.JSONStringer;
import org.json.JSONException;
```

和之前的服务器代码唯一的不同之处在于需要导入 JSON 库。因为在客户端你需要创建 JSON 对象，所以你将导入 `JSONStringer` 对象，以便能够以 OOP 的方式创建 JSON 字符串。`JSONException` 是必需的，因为当你将 JSON 对象转换为字符串时，可能会抛出异常。下面是类中的 `init` 方法。

清单 A.10 客户端 init 方法

```
public Client init()
throws Exception {
  ConnectionFactory factory = new ConnectionFactory();
  factory.setUsername("rpc_user");
  factory.setPassword("rpcme");
  connection = factory.newConnection();
  channel = connection.createChannel();
  return this;
}
```

这里的代码和服务器端的代码很像。你创建了 `ConnectionFactory` 对象，然后设置了用户名和密码。你获取了 `Connection` 实例，并得到信道对象。你将信道和连接对象作为类的成员变量。在获取连接之后，你需要设置消费者。

清单 A.11 客户端消费者设置

```
public Client setupConsumer()
throws Exception {

  replyQueueName = channel.queueDeclare().getQueue();
  consumer = new QueueingConsumer(channel);
  channel.basicConsume(replyQueueName, false, consumer);
  return this;
}
```

该方法的重点在于你声明了一个匿名队列，让 RabbitMQ 为你生成一个队列名。你将队列名存放在成员变量 `replyQueueName` 中。之后，该变量将用于 `reply_to` 的消息属性。

清单 A.12 发送 RPC 请求

```
public String call(String message) throws Exception {
  String response = null;

  channel.basicPublish(                                    ← ❶ 发送RPC请求
    "rpc",
    "ping",
    getRequestProperties(),                                ← ❷ 添加请求属性
    message.getBytes()
  );

  System.out.println("Sent 'ping' RPC call. Waiting for reply...");

  while (true) {                                           ← ❸ 等待RPC回应
    Delivery delivery = consumer.nextDelivery();
    response = new String(delivery.getBody(), "UTF-8");
    break;
  }

  return response;
}
```

call 方法接收一个字符串作为参数，用作发送给服务器的消息载荷。你用字符串 ping 作为路由键，将消息发布到 rpc 交换器上❶。通过调用 getRequestProperties 方法来创建消息属性❷。然后在 while（true）循环中等待回应❸。为了从 RabbitMQ 接收消息，你将采用和服务器端代码相同的技术。在收到消息投递之后，你将消息体转换为字符串，然后返回给调用方。现在让我们看看你如何创建消息属性来将 replyQueueName 包含到 reply_to 消息的基本属性中。

清单 A.13 创建消息的基本属性

```
private BasicProperties
  getRequestProperties() {
    return new BasicProperties
               .Builder()
               .replyTo(replyQueueName)
               .build();
}
```

该方法很简单。你通过使用 builder 技术，链式调用方法设置了需要的每个基本属性，来创建 BasicProperties 对象。虽然在本例中你只设置了 replyTo 属性，但是你也可以通过使用这种方式来设置像 correlationId 或者 deliveryMode 这样的属性。现在让我们看看 main 方法。

清单 A.14 客户端 main 方法

```
public static void main(String[] args) {
  Client client = null;
  String response = null;

  try {
    client = new Client();
    client.init().setupConsumer();
    response = client.call(Client.createRequest());
    System.out.println("RPC Reply --- " + response);
  }
  catch  (Exception e) {
    e.printStackTrace();
  }
  finally {
    if (client!= null) {
      try {
        client.close();
      }
      catch (Exception ignore) {}
    }
  }
}
```

❶ 调用RPC服务器

这里的代码很像初始化服务器的那段代码。有趣的地方在于你如何调用服务器❶。在拥有了消费者实例之后，执行 call 方法，然后等待服务器的响应。注意，对于使用RPC客户端的用户来说，调用本地方法和调用RPC方法之间没有明显差异，因此在这一点上需要注意，因为 RPC 调用要比本地方法慢好几个数量级。还有一点未提到的是 Client.createRequest 代码，它用于生成 JSON 字符串发往服务器。让我们现在就来看看代码吧。

清单 A.15 生成 JSON 请求

```
public static String createRequest()
throws JSONException {
  float epoch = System.currentTimeMillis()/1000;
  JSONStringer msg = new JSONStringer();
return msg
        .object()
        .key("client_name")
        .value("RPC Client 1.0")
        .key("time")
        .value(Float.toString(epoch))
        .endObject().toString();
}
```

首先获取当前 UNIX 时间戳，用于和 RPC 消息一同发送出去。然后，创建

JSONStringer 对象的实例，它提供了 OOP 接口来构建 JSON 对象。就像你看到的那样，你创建了 JSON 对象，并分别设置了 client_name 和 time 属性为 RPC Client 1.0 和 Float.toString(epoch)。

让我们总结一下客户端是如何工作的。首先你需要在类的 main 方法中实例化 RPC 客户端。在有了 Client 实例之后，你可以使用 call 方法来从服务器端得到回应。作为消息发送出去的 JSON 对象是在 createRequest 方法中构建的。在客户端初始化过程中，你在服务器端为客户端声明了队列，并将队列名称存储为对象状态。该队列名称也随着 JSON 对象传送到服务器上，所以服务器才知道往哪里发送回应。一旦得到了服务器的回应，你就将内容打印到控制台上并退出程序。在结束应用程序之前，你需要负责将连接关闭来清理资源。以下是完整的 RPC 客户端代码，包含了 Client.close 方法。

清单 A.16 RPC 客户端完整代码

```
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.QueueingConsumer;
import com.rabbitmq.client.QueueingConsumer.Delivery;
import com.rabbitmq.client.AMQP.BasicProperties;
import org.json.JSONStringer;
import org.json.JSONException;

public class Client {

  private Connection connection;
  private Channel channel;
  private String replyQueueName;
  private QueueingConsumer consumer;

  public Client init()
  throws Exception {
    ConnectionFactory factory = new ConnectionFactory();
    factory.setUsername("rpc_user");
    factory.setPassword("rpcme");
    connection = factory.newConnection();
    channel = connection.createChannel();
    return this;
  }
  public Client setupConsumer()
  throws Exception {
    replyQueueName = channel.queueDeclare().getQueue();
    consumer = new QueueingConsumer(channel);
    channel.basicConsume(replyQueueName, false, consumer);
    return this;
  }
```

```
public String call(String message) throws Exception {
  String response = null;

  channel.basicPublish(
    "rpc",
    "ping",
    getRequestProperties(),
    message.getBytes()
  );
  System.out.println("Sent 'ping' RPC call. Waiting for reply...");

  while (true) {
    Delivery delivery = consumer.nextDelivery();
    response = new String(delivery.getBody(), "UTF-8");
    break;
  }

  return response;
}

public void close() throws Exception {
  connection.close();
}

private BasicProperties
  getRequestProperties() {
    return new BasicProperties
               .Builder()
               .replyTo(replyQueueName)
               .build();
}

public static String createRequest()
throws JSONException {
  float epoch = System.currentTimeMillis()/1000;
  JSONStringer msg = new JSONStringer();
  return msg
          .object()
          .key("client_name")
          .value("RPC Client 1.0")
          .key("time")
          .value(Float.toString(epoch))
          .endObject().toString();
}

public static void main(String[] args) {
  Client client = null;
  String response = null;

  try {
    client = new Client();
    client.init().setupConsumer();
    response = client.call(Client.createRequest());
    System.out.println("RPC Reply --- " + response);
  }
  catch  (Exception e) {
```

```
    e.printStackTrace();
  }
  finally {
    if (client!= null) {
      try {
        client.close();
      }
      catch (Exception ignore) {}
    }
  }
}
```

让我们执行以下命令来编译代码：

```
$ javac -cp ./lib/rabbitmq-client.jar:./lib/json-20090211.jar \
  Client.java
```

如果编译成功的话，那么你就有了新的 Client.class 文件：

```
$ ls Client.class
Client.class
```

现在启动 RabbitMQ 服务器，你就能测试 RPC 客户端和服务器了。记住，你需要和第 4 章相同的设置——你需要将 rpc_user 配置在你的服务器上。

A.3.5 测试 RPC 客户端和服务器

打开两个终端窗口，然后在第一个终端输入以下命令启动服务器：

```
$ java -cp $CP Server
Waiting for RPC calls...
```

现在服务器已经准备好接收客户端的请求了。让我们转到第二个终端窗口，设置我们之前解释过的 CP 变量，然后输入：

```
$ java -cp $CP Client
Sent 'ping' RPC call. Waiting for reply...
RPC Reply --- Pong!1.32087475E
```

你可以看到屏幕上立刻打印出了回应信息。另一方面，在服务器运行的那个终端窗口，你应该看到如下输出：

```
Received API call...replying...
```

通过这个例子，我们完成了 Java 平台的 RabbitMQ 客户端的讲解。作为练习，

请尝试运行 Python RPC 服务器来和 Java 客户端交互，或者反过来测试两种 AMQP 客户端的互操作性。

A.4 总结

当我们开始介绍本附录内容的时候，你也许已经有了一定的 RabbitMQ 基础，但是从 Java 和 .NET 平台上使用这些基础知识可能有些难以捉摸。现在你已经将 Hello World（第 2 章）和告警服务器（第 4 章）示例转换成了 .NET 平台实现，并拥有了一个可以运作的 Java 平台的 RPC 客户端和服务器，它和最初的 Python 实现之间可以完全互操作（第 4 章）。我们希望这段在 Java 和 .NET 平台上使用 Rabbit 的闪电般的学习旅程，能够为你带来必要的思维导图，能够将本书中的所有概念和示例用于你青睐的字节码解释语言。

附录B
在线资源

在本附录中，我们收集了一些有趣的在线资源。当你在寻找 RabbitMQ 相关的信息（不论是支持你喜爱的编程语言的客户端库，还是解决你当前问题的最新消息通信设计模式）时，这些资源都能让你倍感轻松。好吧，实话实说：虽然我们无法照顾到你方方面面的需求，但是我们会尽力列出那些对我们来说有帮助的资源。

B.1 这些站点你应该知道

让我们从回顾一些网站开始。

- RabbitMQ 官方文档——如果你正寻找有关 RabbitMQ 的信息的话，这就是首选之地。自从我们开始编写本书以来，官方网站上的资源已经得到了非常可观的加强。更多信息请参阅：http://www.rabbitmq.com/documentation.html。
- AMQP 参考文档——如果你想更深入地理解 AMQP 的话，则可以访问该协议的官方网站（http://amqp.org/），在那里你可以找到各种版本的规范（http://amqp.org/resources/download）。除了这些链接之外，RabbitMQ 开发团队还创建了 AMQP 快速指引：http://www.rabbitmq.com/amqp-0-9-1-quickref.html。当你想要知道某个 AMQP 方法第四个参数的含义的时候，检索一下这个网页即可。

- 企业集成模式——如果你对消息通信和集成模式感兴趣的话，那么由 Gregor Hohpe 和 Bobby Woolf 一同编写的 *Enterprise Integration Patterns* 一书值得一读。这里我们必须对 AMQP 用户提醒一下：书中的所有示例都是基于诸如 JMS 或者 MSMQ 技术的。你需要稍微转换一下概念。同时，一些模式已经应用在了 RabbitMQ 上。好消息是这本书无须购买。针对模式的叙述和图解是在知识共享署名协议（Creative Commons Attribution License）下发布的，因此你可以在线阅读 http://www.eaipatterns.com/eaipatterns.html。像发布订阅（Publish Subscribe）、竞争消费者（Competing Consumers）以及一些其他模式，我们在第 4 章的示例中曾介绍过。
- Ruby AMQP gem 文档站点——Ruby AMQP gem 站点上有很多关于如何使用 RabbitMQ 和 AMQP 的文档和示例。示例中甚至还有滑稽的兔子画：http://rubyamqp.info/。
- RabbitMQ 开发 RSS 订阅——RabbitMQ Mercurial 库提供了一个有关最新服务器代码变更的 RSS 订阅。如果你想要站在最前沿的话，则这些信息对你来说一定很有意思：http://hg.rabbitmq.com/rabbitmq-server/rss-log。

B.2 博客

- RabbitMQ 官方博客——RabbitMQ 团队在 http://www.rabbitmq.com/blog/ 上维护了一个博客。你可以看到他们在该博客上讨论服务器的新功能开发、提供性能调优提示等。一定要留意这个博客哦。
- Jason's plans——Jason 早在 2009 年的时候就开了自己的博客。那时候我们还在找寻关于如何使用 RabbitMQ 和 AMQP 的示例。他那篇名为 *Rabbits and warrens* 的文章在许多年里一直对人们有启发性。偶尔有人会重新发现它，同时它又会再次出现在 Twitter 上。请参阅这里：http://blogs.digitar.com/jjww/2009/01/rabbits-and-warrens/。
- Alvaro 的博客——Alvaro 维护了一个博客来讨论消息通信和许多其他软件议题。你可以在那里找到诸如用 RabbitMQ 实现 RPC 和 Haskell 的短篇文章：http://videlalvaro.github.com/2010/10/rpc-over-rabbitmq.html。

B.3 AMQP库和相关的OSS项目

- 开发工具和学习资料——RabbitMQ 官方网站有一个开源项目列表，这些项目以某种方式和 RabbitMQ 相关联：http://www.rabbitmq.com/devtools.html。在这里你可以找到从 AMQP 库到 Github 库的各种链接，提供可替代的交换器实现——甚至有社区创建的视频教程！

B.3.1 客户端库

在这里我们专门列出了一些有意思的 AMQP 和 RabbitMQ 库。

- Java——除了官方 Java 客户端外，你还有其他方式来集成 Java 和 RabbitMQ。Spring 框架提供了 Spring AMQP 库：http://www.springsource.org/spring-amqp。Mule ESB 也提供与 AMQP 和 RabbitMQ 的集成。你可以在这里了解更多详情：https://github.com/mulesoft/mule-transport-amqp/blob/master/GUIDE.md。
  最后，最初创建 RabbitMQ 的公司 LShift 针对 Apache Camel 开发了一个实验性质的 RabbitMQ 组件：https://github.com/lshift/camel-rabbitmq。
- .NET/C#——对 C# 来说，有之前附录中提到的官方客户端，而且 Spring AMQP 也提供一个库来和 Spring 集成框架一起使用。该链接和之前 Java 的一样：http://www.springsource.org/spring-amqp。
  除了上述库之外，Mike Hadlow 创建了一个名为 EasyNetQ 的库，位于 https://github.com/mikehadlow/EasyNetQ/wiki/Introduction 上。这个库的目标是："尽可能少的配置（甚至是零配置）"并且"简单的 API"。
- Python——在本书中我们使用了名为 Pika 的库。作为 RabbitMQ 开发团队中的一员，Marek Majkowski 开始研发一个新的名为 Puka 的库。他在下列网址中解释了新库背后的设计原因：http://www.rabbitmq.com/blog/2011/07/08/puka-rethinking-amqp-clients/。
- C——针对 C 语言，有一个名为 `rabbitmq-c` 的库，是由 David Wragg 编写并维护的，同时也对 RabbitMQ 有用。该库也被 C++ 和 Objective-C 的用户包装使用。它位于 RabbitMQ Mercurial 代码库：http://hg.rabbitmq.com/rabbitmq-c/。
- PHP——针对 PHP，我们使用 `php-amqplib` 客户端库，它是纯 PHP 实现。还有一个 AMQP 的 PECL 扩展，你可以在 http://pecl.php.net/package/amqp

上找到。在编写本书的时候，该库正处于大力发展之中。记住，它依赖于rabbitmq-c库。

另一个纯PHP库由于作者的积极开发和文档工作而值得一提。它的名字是amqphp，你可以在https://github.com/braveSirRobin/amqphp找到它。

- Ruby——针对Ruby，你可以尝试我们先前提到的Ruby AMQP Gem（http://rubyamqp.info/）。如果你使用JRuby的话，有一个可替代的库叫作hot_bunnies，你可以在https://github.com/ruby-amqp/hot_bunnies上找到它。好消息是，它是由AMQP Gem的其中一位作者维护的。
- Erlang——Erlang是用来编写RabbitMQ的语言。如果你需要在Erlang上使用RabbitMQ的话，RabbitMQ团队提供了一个客户端库。绝大多数RabbitMQ插件使用该库，所以有可能你正在RabbitMQ下间接地使用它呢。具体使用细节可以在RabbitMQ官方网站上找到：http://www.rabbitmq.com/erlang-client-user-guide.html。
- JavaScript、Node.js和Web消息通信——虽然JavaScript是一种前端语言，但是最近服务器端框架的出现，向人们展示了该语言作为后端编程语言也可以很有趣。Node.js就允许你使用JavaScript来编写服务器端代码，并且你能想象得到用不了多久就会有人为其编写AMQP客户端。其中之一就是node-amqp库，它是由*Scalable Internet Architectures*一书的作者Theo Schlossnagle维护的。更多内容请访问：https://github.com/postwait/node-amqp。

  还有一个叫作Rabbit.js的库，它针对Web实现了多种消息通信模式。它是由RabbitMQ开发团队中的Michael Bridgen创建的。更多详情请访问：https://github.com/squaremo/rabbit.js。
- Clojure——针对Clojure平台有个叫作langohr的库。它的作者Michael Klishin同时也是Ruby AMQP Gem和hot_bunnies库的维护者之一。该库是建立在作者编写Ruby客户端的经验之上的。该库将Java官方库包装为符合Clojure的语言习惯。以下是Github代码库：https://github.com/michaelklishin/langohr。
- Scala——针对Scala，有一个AMQP库可以直接同AKKA可伸缩性（Scalability）框架一同使用。该库的文档可以在其代码库中找到：http://doc.akka.io/docs/akka-modules/1.3.1/modules/amqp.html.。
- Haskell——如果你想要在Haskell上使用RabbitMQ的话，请参阅该amqp包：http://hackage.haskell.org/package/amqp。这里有一份入门指南：http://

videlalvaro.github.com/2010/09/haskell-and-rabbitmq.html。

- NoSQL——RabbitMQ 也和一些 NoSQL 数据库进行了集成；其中最值得注意的是 Jon Brisbin 的作品。他一直致力于将 RabbitMQ 桥接到 Riak。有一个可选的 RabbitMQ 交换器，可以将消息记录到 Riak 中，请参阅 https://github.com/jbrisbin/riak-exchange。同时，如果你想要接收 Riak 数据库中发生更改的通知的话，Jon 编写了一个 Riak post-commit-hook。更多信息请参考 https://github.com/jbrisbin/riak-rabbitmq-commit-hooks。

B.4 讨论和邮件列表

- 邮件列表——如果你想要联系 RabbitMQ 社区和开发者们的话，有几个地方你可以去找找。首先，可以通过官方的邮件列表：http://lists.rabbitmq.com/cgi-bin/mailman/listinfo/rabbitmq-discuss。该列表非常活跃，以至于有些时候你在当天下午就能获得刚提出问题的答案了。如果你使用 RabbitMQ 遇到麻烦的话，这里是寻求帮助最好的地方了。
- IRC 聊天频道——如果你更倾向于现场交流的话，那么可以使用 `#rabbitmq` IRC 聊天频道：http://freenode.net/ network。随便问，这里总是有不少来自社区的成员，他们很愿意为其他人提供帮助。
- Twitter——最后，如果你想要随时了解有关 RabbitMQ 的讨论，则可以在 Twiiter 上订阅 `#rabbitmq`：http://twitter.com/#!/search?q=%23rabbitmq。

B.5 总结

本附录提供了三十多条新资源连接。如果你想要“深挖兔子洞”的话，那么就有一大堆功课要做了。我们提供了有意思的网站和博客中的一系列精选链接，并且为我们提到的每个库都给出了简短的评论。有了 RabbitMQ 和 AMQP，你将不再受限于特定于语言的解决方案。最后，如果你想要联系社区的话，则可以使用传统的 email，直接进行 IRC 交谈，或者通过 Twitter 来实时了解 RabbitMQ 用户身上发生的事情。

附录C

在Windows上安装RabbitMQ

和大多数从 UNIX 起源的程序一样，在 Windows 上安装 RabbitMQ 和我们在第 1 章提到的安装说明有所不同。幸运的是，Rabbit HQ 的团队为我们提供了基于 MSI 的安装包，这使得整个安装过程相对轻松。不过在安装 Rabbit 之前，你必须首先从 http://www.erlang.org/download.html 安装最新版本的 Erlang。通常来讲，erlang.org 上可用的最新版 Erlang 能够和最新版 RabbitMQ 一起工作。那么现在就动手下载 Windows Binary File 吧（见图 C.1）。

在下载 Erlang 安装包之后（在这里是 `otp_win32_R14B04.exe`），进入下载目录并运行（见图 C.2）。

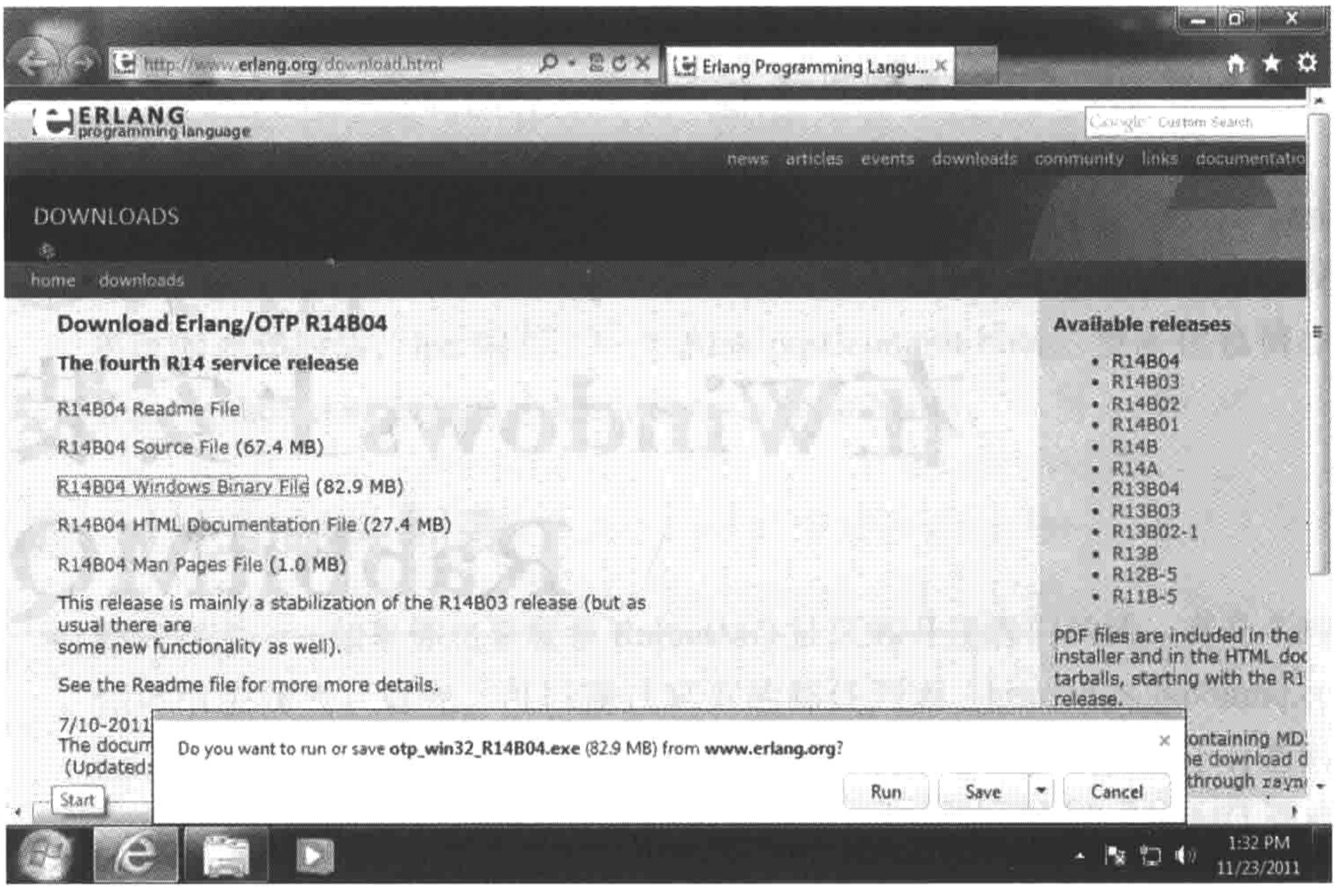

图 C.1 下载 Erlang 的 Windows 安装包

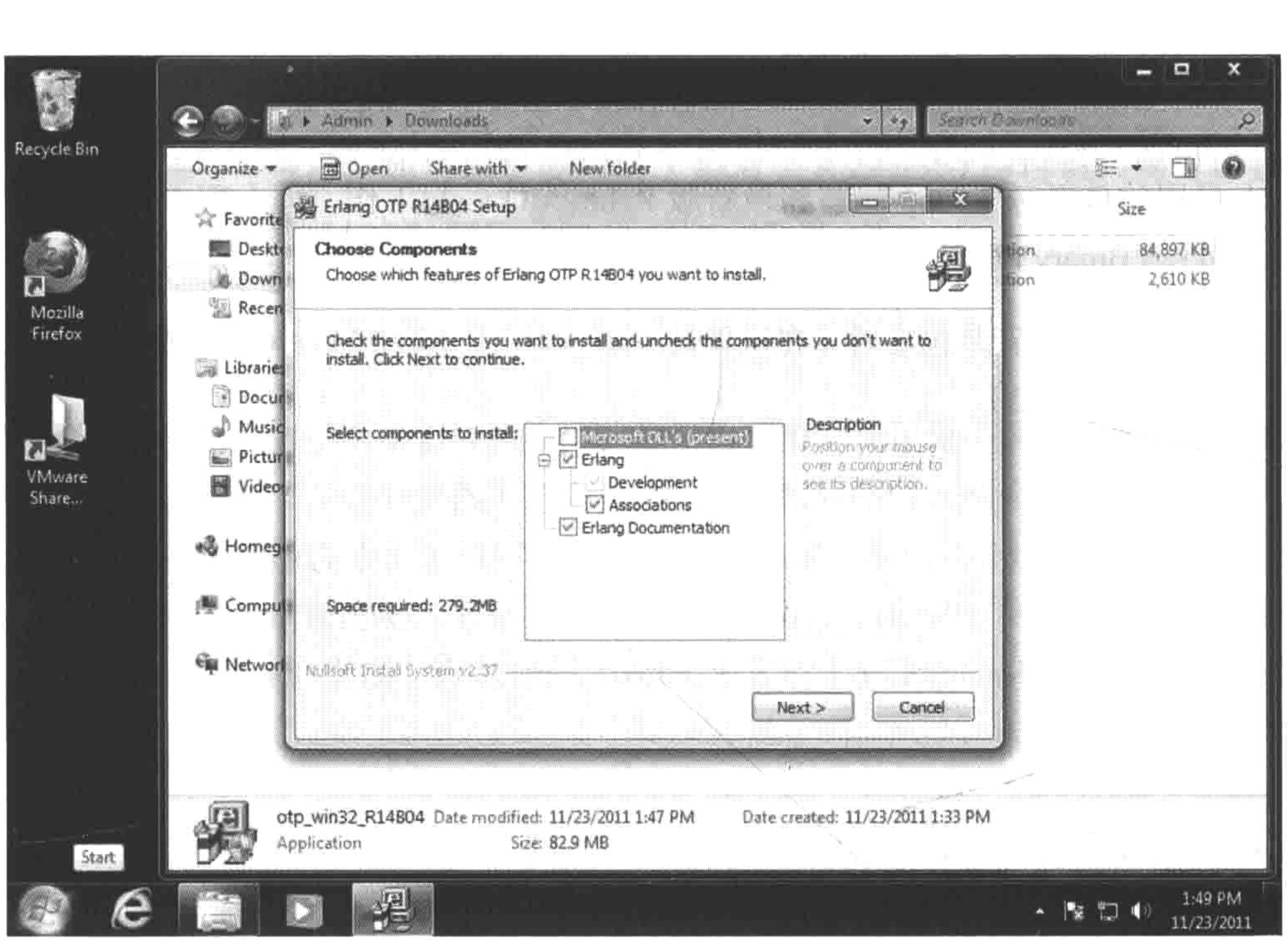

图 C.2 在 Windows 上安装 Erlang

Windows 版 Erlang 安装包的所有默认配置对 Rabbit 来说都没问题，所以在安装向导的所有步骤中都单击 Next（或者 Finish）按钮来完成 Erlang 的默认安装。

下一步，从 http://www.rabbitmq.com/download.html 下载 RabbitMQ 的 Windows 安装包（见图 C.3）。

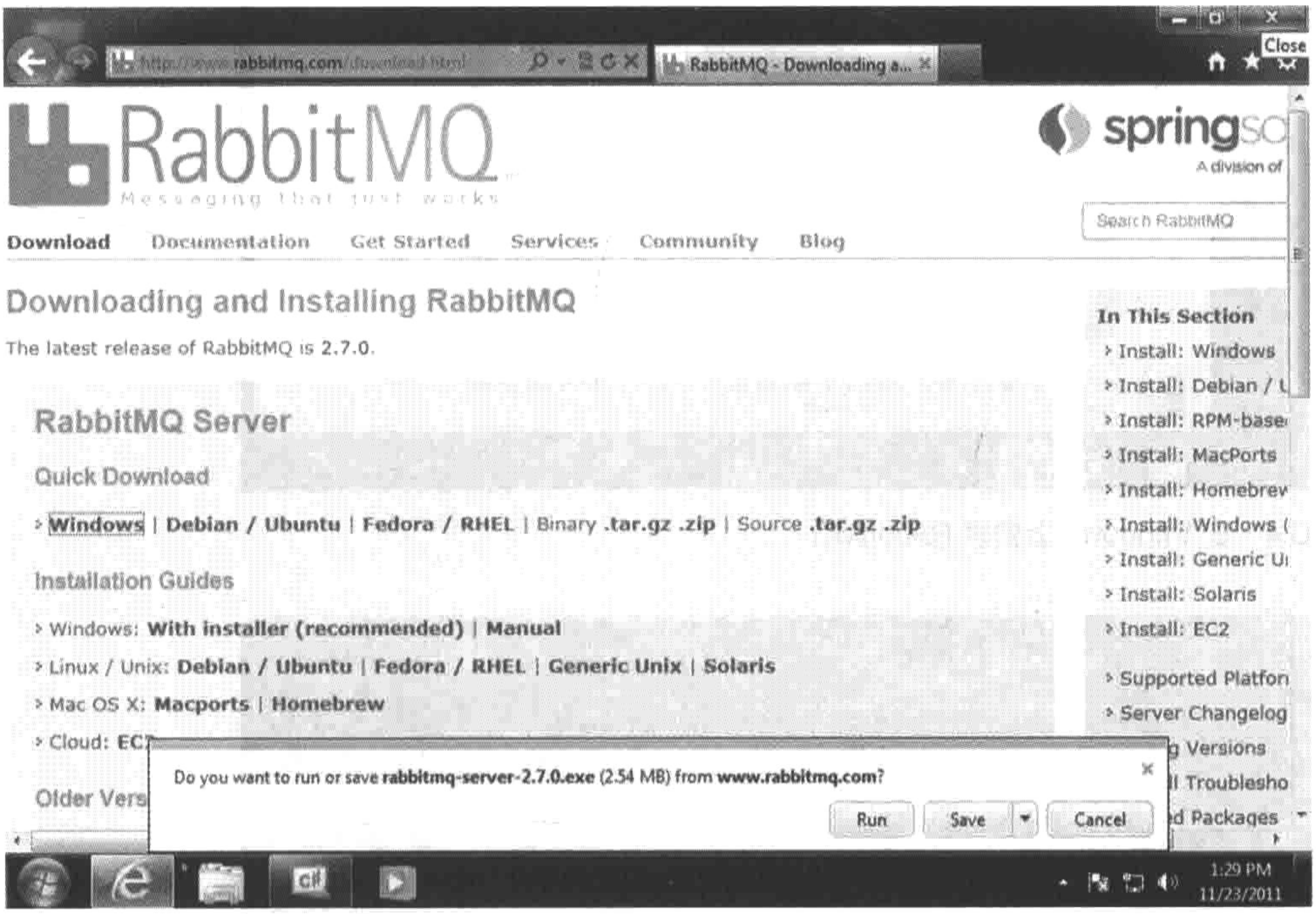

图 C.3　下载 RabbitMQ 的 Windows 安装包

Windows 版的 RabbitMQ 安装包和 Erlang 安装包使用起来同样简单：运行下载好的安装包文件（见图 C.4），然后使用默认配置一路单击，直到安装完成即可。

当安装完成后，你将拥有一个全部功能的 RabbitMQ 拷贝，并以 Windows 服务的方式运行！如果你打开一个命令提示符，运行 `cd C:\Program Files\RabbitMQ Server\rabbitmq_server-2.7.0`，就应该可以看到和其他 RabbitMQ 支持的任何平台一样的目录结构。其中和 UNIX 版 RabbitMQ 的一个不同之处在于 `.\sbin` 子目录下的实用工具是以 `.bat` 结尾的。举例来说，在 Windows 上你需要运行 `.\sbin\rabbitmqctl.bat`，来替代 `.\sbin\rabbitmqctl`。而且，通过 Erlang 来和 Rabbit 通信的任何实用工具（例如 `rabbitmqctl.bat`）在第一次运行的时候，你将收到来自 Windows 防火墙的警告（见图 C.5），询问你是否允许 `erl.exe` 访问网络。直接单击允许 `erl.exe` 访问“私有”网络。

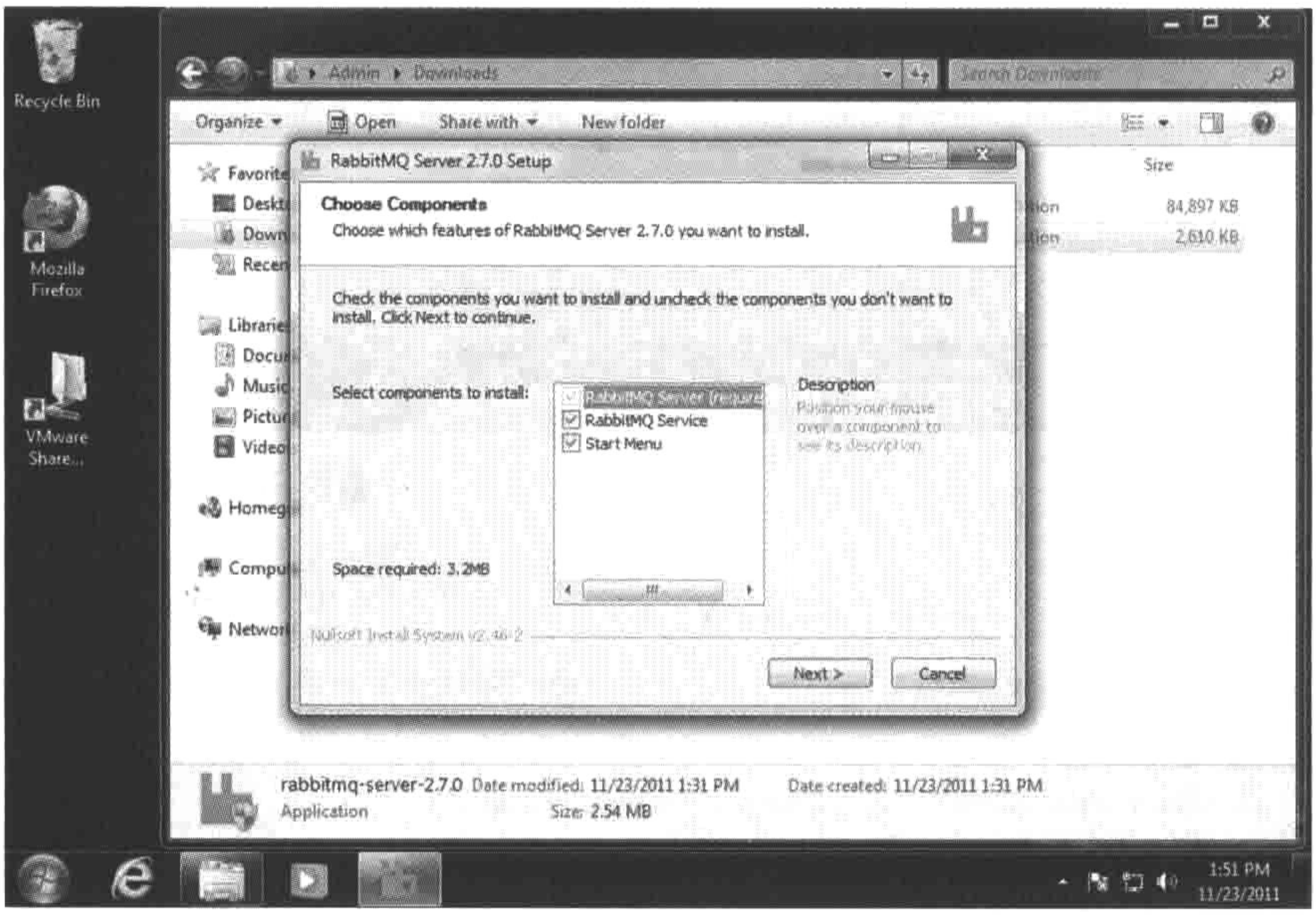

图 C.4 在 Windows 上安装 RabbitMQ

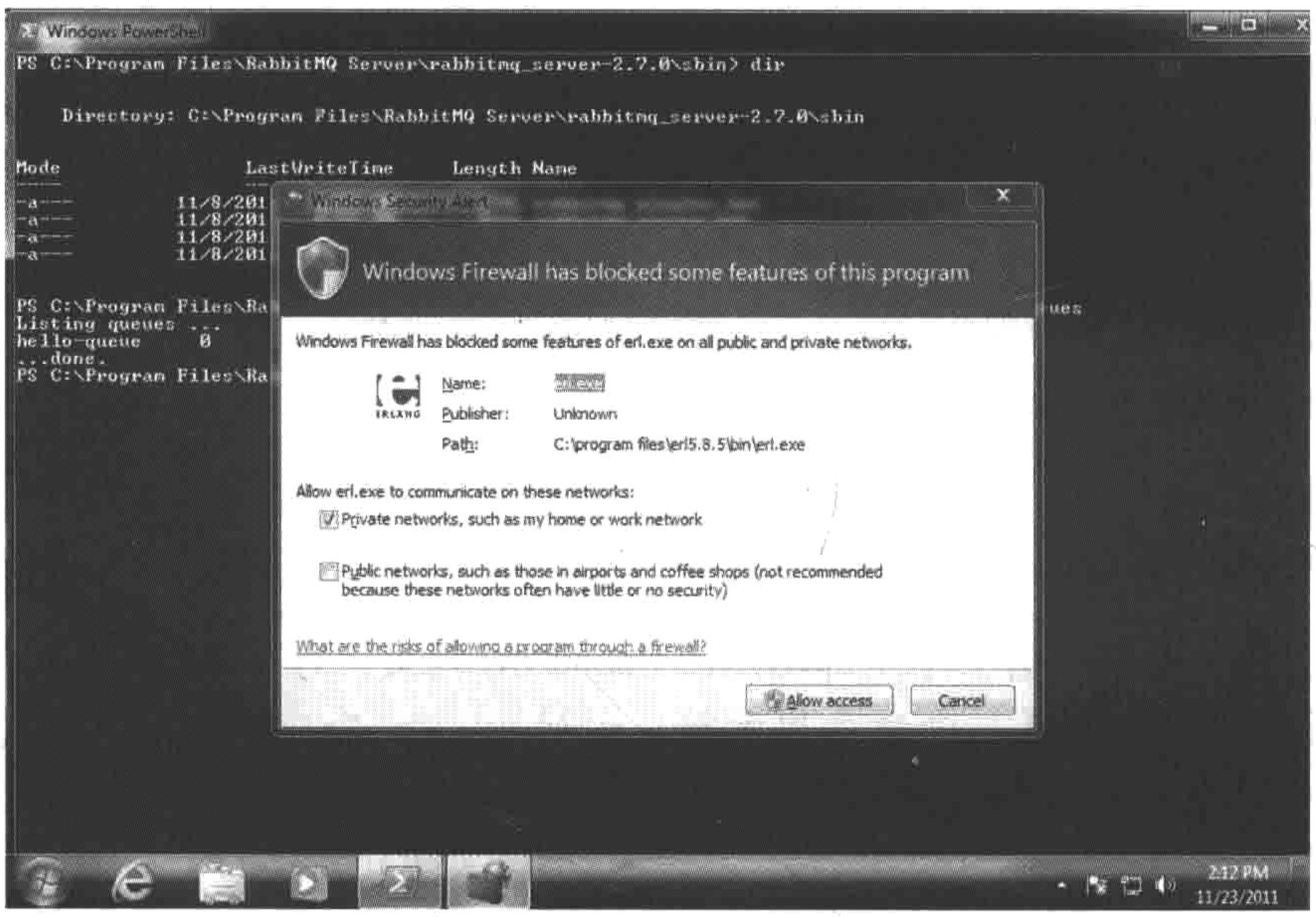

图 C.5 设置 Windows 防火墙以允许 erl.exe 访问网络

以上就是所有内容了！你现在在 Windows 系统上拥有了 RabbitMQ 全部功能的拷贝了！本书的所有示例均能在 Windows 版的 Rabbit 上（特别是附录 A 中的 .NET 示例）和在 UNIX 版上一样运行。这就是 Rabbit 的美妙之处！

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

RabbitMQ实战
高效部署分布式消息队列

绝大多数应用程序的核心都有一个虚拟交换机，用于在服务器、程序和服务之间快速传递消息。RabbitMQ是一个高效且易于部署的消息队列服务器，无论是网络初创公司还是大型企业系统，它均可以轻而易举地处理各种情形下的消息通信。

本书能够教会你如何使用RabbitMQ消息通信服务器，构建和管理用不同语言编写的可伸缩应用。这很容易上手。你将学习消息队列的工作原理以及RabbitMQ是如何适用的。然后，你将通过许多案例来探索实践当中的可扩展性和互操作性问题。最后，学习如何将Rabbit打造成一部运作良好的机器，以应对24×7×365的无间断工作环境。

本书内容包括

- 学习基本消息通信设计模式
- 使用模式来满足按需扩展
- 整合PHP编写的前端和用任何语言编写的后端
- 在30分钟内实现发布/订阅的告警服务
- 配置RabbitMQ内建集群
- 监控、管理、扩展并调试RabbitMQ

本书是为那些熟悉Python、PHP、Java、.NET或者任何其他现代编程语言的开发者编写的。无须任何RabbitMQ经验。

Alvaro Videla是一名专门从事基于MQ应用程序的开发者和架构师。

Jason J.W. Williams是消息通信服务供应商DigiTar公司的CTO，他领导着设计与开发工作。

两位专家在这本杰作中为我们分享了多年来运行大规模RabbitMQ系统的经验。

——Alexis Richardson，本书序作者

写得很棒，充满智慧，简单易懂。

——Karsten Strøbæk，微软

这是RabbitMQ大全，当中包含了各种有广度、有深度的案例。

——Patrick Lemiuex，来自Voxel Internap

本书将带你领略消息通信的美妙世界。

——David Dossot，*Mule in Action*的合著者

MANNING

策划编辑：张春雨
责任编辑：李云静
封面设计：李　玲

上架建议：网站开发/运维

ISBN 978-7-121-27164-9
9 787121 271649 >
定价：75.00元

[General Information]
书名=RabbitMQ实战　高效部署分布式消息队列
作者=（美）维德拉，（美）威廉姆斯著
页数=310
SS号=13888679
DX号=
出版日期=2015.10
出版社=北京电子工业出版社