高级消息队列协议

协议说明书

版本 0-9-1，2008.11.13

通用消息标准

目录

1 概述

1.1 本文目的

本文定义了一种网络协议，即高级消息队列协议（AMQP），该协议使得遵从该协议的客户端应用程序和消息中间件服务器之间能够互相通信。我们面对在该领域有经验的技术读者，提供了足够的规范和指南，一个熟练的工程师可以根据这些文档在任何现代编程语言或硬件平台构建合适的解决方案。

1.2 概要

1.2.1 为什么是 AMQP

AMQP 在相符合的客户端和消息中间件服务器（也称“代理”），实现全功能之间的互操作性。

我们的目标是能够发展并全行业使用的标准消息中间件技术，以便降低企业和系统集成的成本，并且向大众提供工业级的集成服务。我们旨在通过 AMQP 消息中间件能力最终将进入网络本身，并且通过消息中间件的普适性能够开发出各种有用的应用程序。

1.2.2AMQP 的范围

为了实现消息中间件的互操作性，需要充分定义网络协议和服务端服务的的功能语义。因此，AMQP 定义了如下网络协议和服务端服务：

- 一套明确的消息交换能力，也称为“高级消息队列协议模型”（AMQP 模型）。AMQP 模型由一套经过代理服务路由和存储消息的组件以及一套将这些组件联系在一起的规则所组成。
- 一个网络线级协议，可以让客户端和消息代理对话，与 AMQP 模型实现交互。

可以部分说明 AMQP 协议规格中的服务语义，但是我们相信明确的语义描述有助于理解协议。

1.2.3 高级消息队列模型（AMQP 模型）

我们需要明确地定义服务器的语义，因为互操作性要求所有给定的服务器实现都需要保持这些语义的一致性。因此 AMQP 模型明确规定了一套模块化的组件以及连接这些组件的标准规则。在服务器中，三个主要类型的组件被连接到处理链中，用来创建所需的功能：

- “交换（exchange）”接收发布应用程序发送的消息，根据任意的标准（通常是消息属性或内容），将这些消息路由到“消息队列（message queues）”。
- “消息队列（message queue）”存储消息，直到这些消息被一个消费客户端应用程序（或多个应用程序）安全地处理完为止。

- “绑定（binding）”定义了消息队列和交换之间的关联，并且提供了消息路由标准。

使用此模型，我们可以很容易模拟出存储转发队列和主题订阅这些经典的面向消息的中间件概念。我们还可以表达较少的概念，如基于内容的路由、工作负载分布和按需消息队列。

大体上来讲，一个 AMQP 服务器类似于邮件服务器，每个交换相当于消息传输代理，每个消息队列相当于邮箱。绑定定义了每一个传输代理中的路由表。发布者向单个传输代理发送消息，然后将消息路由到邮箱。消费者从有相中取出消息。相比之下，在很多类 AMQP 中间件系统，发布者直接将消息发送给单个邮箱（就存储转发来说）或者邮件列表（就主题订阅来说）。

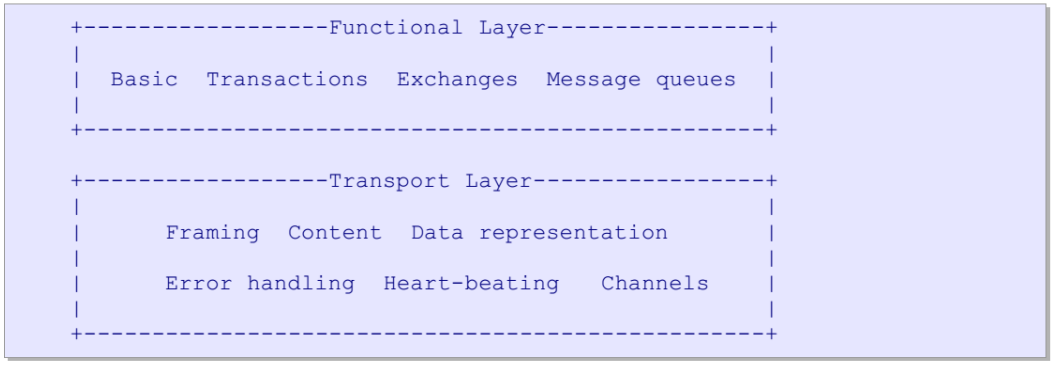
区别在于，消息队列连接到交换的规则受制于系统架构（非嵌入在代码中），这就使得很多有趣的事成为可能，比如定义一条规则，“将所有包含这样那样的消息头的消息都复制一份放置到此消息队列中”。

AMQP 模型的设计是基于下面需求驱动的：

- 支持主要消息传递产品的语义要求
- 提供主要消息传递产品的性能要求
- 通过 AMQP 协议，允许服务器的特定语义可以由应用程序编程
- 灵活、可扩展以及简单

1.2.4 高级消息队列协议（AMQP）

AMQP 协议是具备现代特征的二进制协议：多通道、协商式、异步、安全、跨平台、中立、高效。AMQP 通常被划分为两层：



功能层定义了一系列的命令（分组到功能逻辑类），代表应用程序做有用的工作。

传输层负责将这些方法从应用程序搬运到服务器并返回，同时处理通道复用，帧同步，内容编码，心跳检测，数据表示和错误处理。

在不改变协议可视功能的前提下，可使用任意的传输协议来替换传输层。也可以将同一个传输层用于不同的高级协议。

AMQP 模型的设计是基于下面需求驱动的：

- 保证一致性实现之间的互操作性
- 提供服务质量的显示控制
- 一贯的、明确的命名
- 通过协议允许服务器完全可配置
- 使用命令标记法可轻易地映射到应用程序级别的 API
- 每个操作只做自己的事

AMQP 传输层的设计师基于以下需求驱动的，不分先后次序：

- 使用能够快速打包和解包的二进制编码来保证数据的紧凑性
- 处理任意大小的消息（实际上有限制）
- 单连接上携带多个通道
- 长时间生存，没有显著的内建限制
- 允许异步流水线命令
- 易于扩展，处理新的和变化的需求
- 高版本向前兼容性
- 使用强断言模型来保证应用程序的可修复性
- 保持编程语言的中立性
- 适合代码生成过程

1.2.5 部署规模

AMQP 范围涵盖不同等级规模，大致如下：

- 开发人员/临时使用：1 台服务器，1 个用户，10 个消息队列，每秒 1 个消息
- 产品化应用程序：2 台服务器，10-100 个用户，10-50 个消息队列，每秒 10 个消息（3.6 万个消息/每小时）
- 部门任务关键应用： 4 台服务器，100-500 个用户，50-100 个消息队列，每秒 100 个消息（36 万个消息/每小时）
- 区域任务关键应用：16 台服务器，500-2000 个用户，100-500 个消息队列，每秒 1000 个消息（360 万个消息/每小时）
- 全球任务关键应用：64 台服务器，2000-10000 个用户，500-1000 个消息队列，每秒 10000 个消息（3600 万个消息/每小时）
- 市场数据（交易）：200 台服务器，5000 个用户，10000 个主题，每秒 100000 个消息（3.6 亿个消息/每小时）

规模越大, 消息传输延迟也越重要。例如, 市场数据很快就变得一文不值。实现可以通过提供不同的服务质量或可管理性能力区分对待, 但必须完全符合本规范。

1.2.6 功能范围

我们想要支持多种消息传递架构：

- 多写者和单读者来存储转发
- 多写者和多读者来工作负载分发
- 多写者和多读者来发布订阅
- 多写者和多读者来基于内容路由
- 多写者和多读者来进行队列文件传输
- 两个节点之间进行点对点连接
- 多源和多读者来市场数据分发

1.3 文档组织

文档分成五个章节，其中大部分设计为可按你的兴趣来独立阅读：

1. “概述”（本章），读本章节来了解介绍。
2. “总体架构”，我们所描述的架构和 AMQP 总体设计。此章节的目的是帮助系统架构师了解 AMQP 如何工作的。
3. “功能说明”，在我们定义了应用程序如何与 AMQP 工作。此章节包括一个可读性的讨论，其次是每个协议命令的详细规范，以作为实施者参考。在阅读此章节之前，您应该阅读总体架构。
4. “技术说明”，我们在其中定义 AMQP 传输层是如何工作的。此章节包括一个简短的讨论，其次是线路级结构的具体规格，以作为实施者参考。如果您想了解有线路级协议是如何工作的（但不是它的用途），您可以自己阅读此章节。

1.4 约定

1.4.1 实施指南

- 我们使用了 IETF　RFC 2119 中定义的术语：MUST，MUST NOT，SHOULD，SHOULD NOT 和 MAY。
- 当讨论符合 AMQP 服务端的特定行为时，我们使用术语“the server”。
- 当讨论符合 AMQP 客户端的特定行为时，我们使用术语“the client”。
- 我们使用术语“the peer”来代表服务端或客户端。
- 除非另有说明，所有数值均为十进制。
- 协议常量显示为大写名称。AMQP 实现应该使用这些名称的定义和源代码和文档使用常数。
- 属性名、方法参数和帧字段显示为小写的名字。AMQP 实现应该始终在源代码和文档使用这些名称。
- AMQP 中的字符串是区分大小写的。例如，“amqp.Direct”与“amqp.direct”时两个不同的交换。

1.4.2 版本号

AMQP 版本使用两个或三个数字进行表示—主版本号，次版本号以及可选的修订版本号。习惯上，版本号表示为：主版本号-次版本号[-修订版本号]或主版本号.次版本号[.修订版本号]：

- 官方说明中，主版本号、次版本号和修订版本号采用 0 到 99 的数字。
- 主版本号、次版本号和修订版本号中 100 及其以上的数字保留用于内部测试和开发。
- 版本号表示语法和语义互操作性。
- 版本号 0-9-1 表示主版本号=0，次版本号=9，修订版本号=1。
- 版本号 1.1 将表示为主版本号=1，次版本号=1，修订版本号=0。“AMQP/1.1”等价于“AMQP/1.1.0”或“AMQP/1-1-0”。

1.4.3 技术术语

这些术语在本文中有特殊的意义：

- AMQP 命令架构（AMQP command architecture）：用于在 AMQP 模型架构上执行操作的线级协议命令
- AMQP 模型架构（AMQP model architecture）：代表关键实体和语义的逻辑框架，它必须对兼容 AMQP 实现的服务器可用，使得服务器的状态可以通过客户端按本规范中定义的语义来实现。
- 连接（Connection）：网络连接，例如：一个 TCP/IP 套接字连接。
- 通道（Channel）：两个 AMQP 节点之间的双向通信流。通道是多路复用的，因此单个网络连接可以支撑多通道
- 客户端（Client）：AMQP 连接或通道的发起者。AMQP 是非对称的。客户端生产和消费消息，而服务端入队和路由消息。
- 服务端（Server）：服务端进程接受客户端的连接，实现 AMQP 消息队列和路由功能。也称为“代理”
- 点（Peer）：AMQP 连接中的任意一方。AMQP 连接涉及两个点（一个是客户端，一个是服务端）。
- 帧（Frame）：一个正式定义的连接数据包。在连接中，帧总是作为连续读和写的一个单位。
- 协议类（Protocol class）：用于处理特定类型功能的 AMQP 命令集合。
- 方法（Method）：用于点对点之间的一个特定类型的 AMQP 命令帧。
- 内容（Content）：从客户端到服务端，从服务端到客户端的应用程序数据。这个术语是“消息”的同义词。
- 内容头（Content header）：描述内容属性的特定类型帧。
- 内容体（Content body）：包含原始应用程序数据的特定类型帧。内容体帧完全不透明—服务端不以任何方式检查或修改这些内容。
- 消息（Message）：与“内容”（Content）同义。
- 交换（Exchange）：服务端中接收来自生产者应用程序的消息的实体，并可以将这些消息路由路由到服务端中的消息队列。
- 交换类型（Exchange type）：一种特殊交换模型的算法与实现。相比之下，“交换实例”，是接收和路由服务端内消息的实体。
- 消息队列（Message queue）：保存消息并将消息转发给消费者应用程序的命名实体。
- 绑定（Binding）：用于创建消息队列和交换器绑定关系的实体。
- 路由键（Routing key）：交换器用来决定如何路由一个特定消息的虚拟地址。
- 持久化（Durable）：可在服务端重启时恢复的服务端资源
- 临时性（Transient）：在服务端重启后擦除或重置的服务端资源
- 持续性（Persistent）：服务器存储在可靠磁盘存储中的消息，在服务器重新启动后不丢失。
- 消费者（Consumer）：从消息队列中请求消息的客户端应用程序。
- 生产者（Producer）：发布消息到交换器的客户端应用程序。
- 虚拟主机（Virtual host）：交换器、消息队列以及相关对象的集合。虚拟主机是共享同一个身份验证和加密环境的独立服务端域。
- 断言（Assertion）：一个必须为 true 且可继续执行的条件。
- 异常（Exception）：一个失败的断言，可通过关闭通道或连接来处理。

在 AMQP 中这些没有特殊含义：

- 主题（Topic）：通常是分发消息的一种手段。AMQP 使用一个或多个交换类型来实现主题。

- 订阅（Subscription）：通常是从主题中接收数据的请求，AMQP 以消息队列和绑定的方式来实现订阅。
- 服务（Service）：通常与服务端一个含义。AMQP 使用“服务端”来遵循 IETF 标准命名。
- 代理（Broker）：通常与服务端一个含义。AMQP 使用术语“客户端”和“服务端”来遵循 IETF 标准命名。
- 路由（Router）：有时用来描述交换器的行为。交换器也可以作为消息的终点，“路由”在网络领域中具有特殊的意义，因此 AMQP 不使用它。

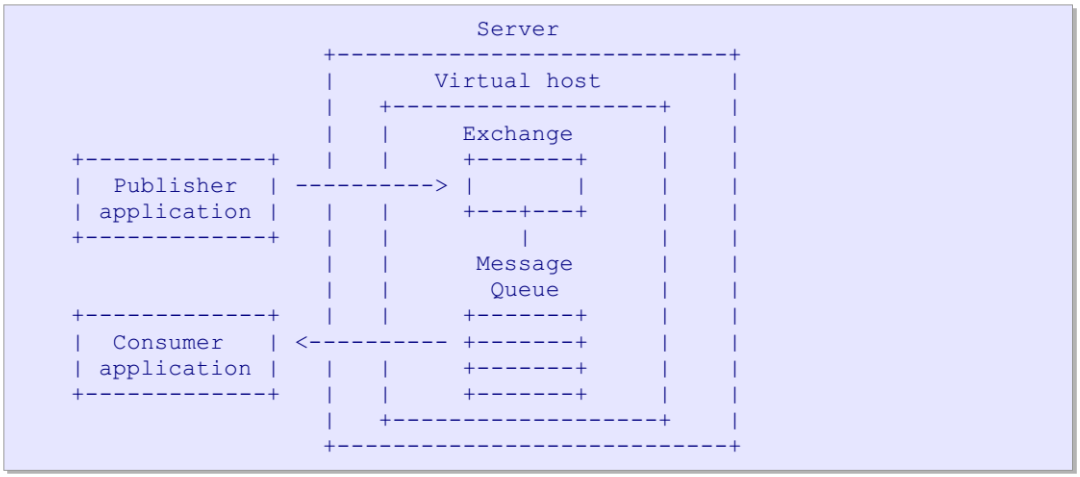
2 总体架构

2.1AMQP 模型架构

本节讲解必须标准化服务端语义，以保证与 AMQP 实现的互操作性。

2.1.1 主要实体

下图显示了整体 AMQP 模型：



我们可以概述下中间件服务器是什么：它是一个接收消息的数据服务器，并做两个主要事情，根据任意标准将消息路由到不同的消费者，当消费者不能足够快速地消费消息时，将消息缓存在内存或磁盘上。

在 AMQP 之前的服务器，这些任务是由实现了特定路由类型和缓冲的单片引擎完成的。AMQP 模型采取较小的、模块化的组件，同时结合更多样和稳健的方法。它首先将这些任务划分为两个不同的角色：

- 交换器，接收来自生产者的消息并将它们路由到消息队列中。
- 消息队列，保存消息并转发它们给消费者应用程序。

交换器和消息队列之间有一个明确的接口，称为“绑定”，稍后我们将介绍它。AMQP 通过两个主要方面来提供运行时可编程语义：

1. 运行时通过协议创建任意交换器和消息队列类型的能力（有些在标准中定义，但是可以添加其他作为服务器扩展）。
2. 运行时通过协议连接交换器和消息队列来创建任何所需的消息处理系统的能力。

2.1.1.1 消息队列

消息队列将消息存储在内存或磁盘中，并将它们依次投递给一个或多个消费者应用程序。消息队列时消息存储和分发的实体。每个消息队列是完全独立的，并且是一个相当聪明的对象。

消息队列有很多属性：私有的或共享的，持久的或临时的，客户端命名或服务端命名的等等。通过选择所需属性，我们可以使用消息队列来实现传统的中间件实体，如：

- 共享存储转发队列，它可以持有消息，并在 round-robin 方式在消费者之间分发消息。存储转发队列通常在多个消费者之间是持久的，共享的。
- 私有应答队列，它可以持有消息，并把消息转发给单个消费者。应答队列通常对某个消费者来说是临时的，服务端命名的，私有的。
- 私有订阅队列，它可以持有来自不同订阅源的消息，并将它们转发给单个消费者。

订阅队列通常对于某个消费者来说是临时的，服务端命名的，私有的。在 AMQP 中这些类别是没有正式定义：它们是如何使用消息队列的示例。创建如持久的共享的订阅队列这样新的实体是没什么意义的。

2.1.1.2 交换器

交换器接收来自生产者应用程序的消息，并将它们按照事先安排的标准路由到消息队列中。这些标准也称“绑定”。交换器是一个匹配和路由的引擎。也就是说，它们会检查消息，并使用它们的绑定表来决定如何将这些消息转发到消息队列中或者其他交换器中。交换器从不存储消息。

术语“交换器”用来表示一类算法或算法实例。

AMQP 定义了许多标准的交换器类型，它们覆盖了常见的消息路由分发的基本类型。AMQP 服务器将这些交换器的默认实例。应用程序使用 AMQP 还可以创建自己的交易实例。交换类型被命名为应用程序，它们创建自己的交换可以告诉服务器要使用什么交换类型。还指定 Exchange 实例，以便应用程序可以指定如何绑定队列和发布消息。使用 AMQP 的应用程序还可以创建自己的交换器实例。交换器类型可以被命名，这样应用程序创建自己的交换器可以告诉服务端他们使用的交换器类型。交换器实例也可以被命名，这样应用程序可以指定如何绑定队列和发布消息。

交换器能不仅仅是路由消息。它们可以作为服务器内的智能代理，按需接收消息和生产消息。交换器概念的目的是定义一套模型，使得可以在一个合理的标准化下扩展 AMQP 服务器，因为可扩展性会对互操作性有所影响。

2.1.1.3 路由键

在一般情况下，交换器会检查消息的属性，如报文头字段以及报文体内容，并且使用这些和可能的其他源的数据来决定如何路由消息。

在大多数简单情况下，交换器会检查单键字段，我们称之为“路由键”。路由键是一个虚拟地址，交换器会使用这个虚拟地址来决定如何路由消息。

对于点对点的路由，路由键通常是消息队列的名字。

对于发布订阅的路由，路由键通常是主题层次值。

在更复杂的情况下，路由键可以是消息头字段和/或消息内容的组合体。

2.1.1.4 类似电子邮件

如果我们做一个类似的电子邮件系统，我们可以看到 AMQP 概念不是全新的：

- AMQP 消息类似于电子邮件消息
- 消息队列类似邮箱
- 消费者类似可读取和删除电子邮件的邮件客户端
- 交换器类似一个 MTA（邮件传输代理），它会检查邮件并基于路由键和表决定何如将电子邮件发送到一个或多个邮箱中
- 路由关键字对应一个电子邮件中的主送或抄送或密送地址，并不包含服务端信息（路由完全是 AMQP 服务器内部的行为）
- 每个交换器实例类似独立的 MTA 过程，用于处理一些电子邮件子域名或特定类型的电子邮件传输
- 绑定类似 MTA 路由表中的实体

AMQP 的强大来自于创建队列（邮箱）、交换器（MTA 过程）、和绑定（路由实体）的能力，在运行时，将这些连接在一起，这远远超出了简单的主送地址到邮箱名字的映射。

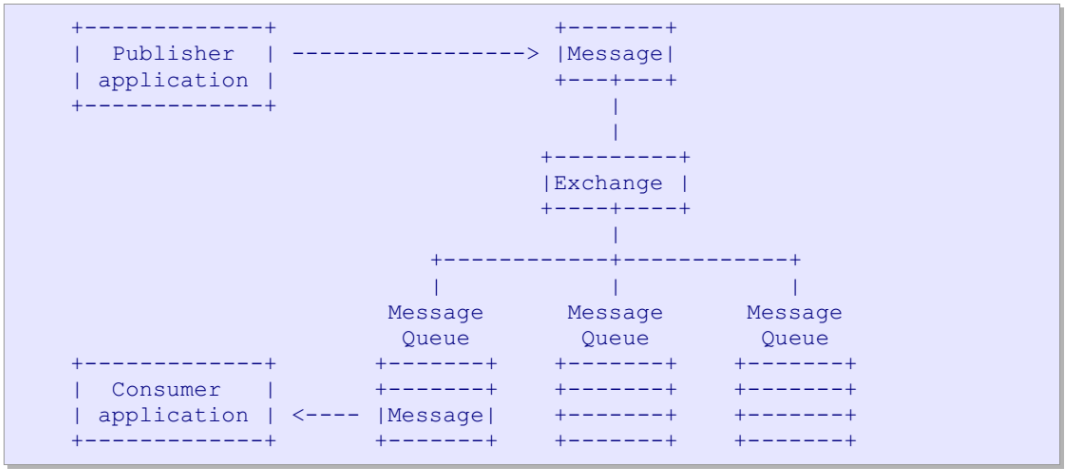
我们不应该把电子邮件 AMQP 类比得太远：它们之间有根本上的区别。AMQP 面临的挑战是要路由和存储在服务器中的消息，或按 SMTP（IETF RFC 821）的说法，称其为“自治系统”。相比之下，电子邮件中的挑战是在自治系统之间如何路由消息。

在一个服务器内部和多个服务器之间的路由是不同的问题，并有不同的解决方案，如果只是出于老套的原因，例如保持透明的性能。

在多个 AMQP 服务器（属于不同的实体之间）路由时，必须建立明确的桥梁，为达到在多个独立实体之间传输消息的目的，一个 AMQP 服务器作为另一个 AMQP 服务器的客户端。这种工作方式很适合需要使用 AMQP 的业务类型，因为这些桥梁被用来支持业务流程、合同义务和安全问题。

2.1.2 消息流

下图显示了通过 AMQP 模型服务器的消息流：



2.1.2.1 消息生命周期

一个 AMQP 消息由一组属性和不透明的内容组成。

一个新消息是由生产者应用程序通过使用 AMQP 客户端 API 来创建的。生产者将内容附着在消息中，并可能设置一些消息属性。生产者使用路由信息来标记消息，其表面上类似于地址，但几乎可以创建任何模式。然后，生产者将消息发送到服务端中的交换器。

当消息到达服务端时，交换器通常会将消息路由到一组存在服务器上的消息队列中。如果消息是不可路由的，交换器可能默默地丢弃或者将消息返回给生产者。生产者可以选择如何处理不可路由的消息。

单个消息可存在于多个消息队列中。服务端可以不同方式进行处理，如通过拷贝消息，通过使用引用计数等。这不影响互操作性。然而，当一个消息被路由到多个消息队列时，它在每个消息队列上都是一样的。没有独特的标识符来区分不同的副本。

当消息到达消息队列时，消息队列会通过 AMQP 协议，立即尝试将消息传递给消费者应用程序。如果不行，消息队列会存储消息（按发布者要求存储在内存或磁盘中），并等待消费者准备好。如果没有消费者，消息队列可能通过 AMQP 协议将消息返回给生产者（再次地，如果生产者对此有要求的话）。

当消息队列能够把消息投递给消费者时，它会从内部缓冲区中删除消息。这有可能立即发生，可有可能在消费者应答它已成功处理消息之后。消费者可选择如何以及何时来应答消息。消费者同样可以拒绝消息（否定应答）。

生产者消息和消费者应答可以组成事务。当一个应用程序同时扮演两种角色时，往往它会做混合工作：发送消息和发送应答，然后提交和回滚事务。

消息从服务端投递给消费者是非事务的，它只能通过消息应答来处理。

2.1.2.2 生产者能看到什么

通过与电子邮件系统的类比，我们可以看到生产者不能直接向消息队列发送消息。如果允许这么做，将会破坏 AMQP 模型中的抽象。这就像允许电子邮件绕过 MTA 路由表，直接发送到邮箱一样。例如，这将不可能插入中间过滤和处理、垃圾邮件检测。

AMQP 模型使用与电子邮件系统一样的准则：所有消息都发向单点、交换器或 MTA，然后它根据隐藏在发送者的规则和信息来检查消息，并将消息路由到落脚点（对于发送者来说，

信息仍然是隐藏的）。

2.1.2.3 消费者能看到什么

当我们站在消费者角度来看与电子邮件系统的类比时，这就开始崩溃了。电子邮件客户端是被动的-它们可以读取它们的邮箱，但是它们不能对邮箱的填充产生任何影响。AMQP 消费者同样是被动的，就像电子邮件客户端一样。也就是说，我们可以编写一个应用程序，它期望一个特定的消息队列准备就绪，并且只需消息队列中的消息。

此外，我们也允许 AMQP 客户端应用程序执行下面的操作：

- 创建或销毁消息队列；
- 通过绑定来定义这些消息队列的填充方式；
- 选择不同能够完全改变路由语义的交换器；

通过 AMQP 协议，这有点像电子邮件系统能做的：

- 创建新邮箱；
- 告诉 MTA 待特定头字段的消息都可以拷贝到这个邮箱中；
- 完全改变电子邮件系统解释地址和其他消息头

我们看到 AMQP 更像是一种连接组件的语言而非一个系统。这是我们目标的一部分，通过 AMQP 协议使服务器行为可编程化。

2.1.2.4 自动化模式

大多数集成架构不需要这种复杂程度。就像业余摄影师一样，大多数 AMQP 用户需要一个傻瓜式的模式。AMQP 通过两个简化概念提供了这种模式：

- 针对消息生产者的默认交换器
- 基于路由键和消息队列名称之间的匹配来选择消息的消息队列的默认绑定。

实际上，给予适当的权限，默认的绑定允许生产者将消息直接发送到消息队列—它模拟了传统中间件简单的“发送到目的地”的寻址方案。

默认绑定不会阻止消息队列以更复杂的方式使用。然而它使得用户在不需要了解交换器和绑定如何工作的情况下，就可以使用 AMQP。

2.1.3 交换器

2.1.3.1 交换器类型

每个交换器类型实现了特定的路由算法。这里有很多标准的交换器类型（将在“功能规格”章节中进行讲解），但有两点特别重要：

- 基于路由键来路由的“direct”交换器类型。默认交换器是“direct”交换器
- 基于路由模式来路由的“topic”交换器类型

服务器会在启动时会以一些总所周知的名字创建一系列交换器，包括“direct”和“topic”的交换器，并且客户端应用程序可能会依赖于此。

2.1.3.2 交换器生命周期

每个 AMQP 服务器都会预先创建一些交换器（实例）。这些交换器在服务器启动时就存在了，不能被销毁。AMQP 应用程序同样可以创建它们自己的交换器。AMQP 不会使用诸如“create”的方法，它使用“declare”的方法，其意义在于：“如果不存在就创建，否则继续”。这对于应用程序为了私有使用而创建交换器，并在完成时进行销毁时合理的。AMQP 提供了方法来销毁交换器，但一般说来，应用程序不会这样做。在本章我们的例子中，我们假设交换器已经在服务器启动时创建过了。我们就不会展示应用程序声明它自己的交换器。

2.1.4 消息队列

2.1.4.1 消息队列属性

当客户端应用程序创建了消息队列时，它可以选择一些重要的属性：

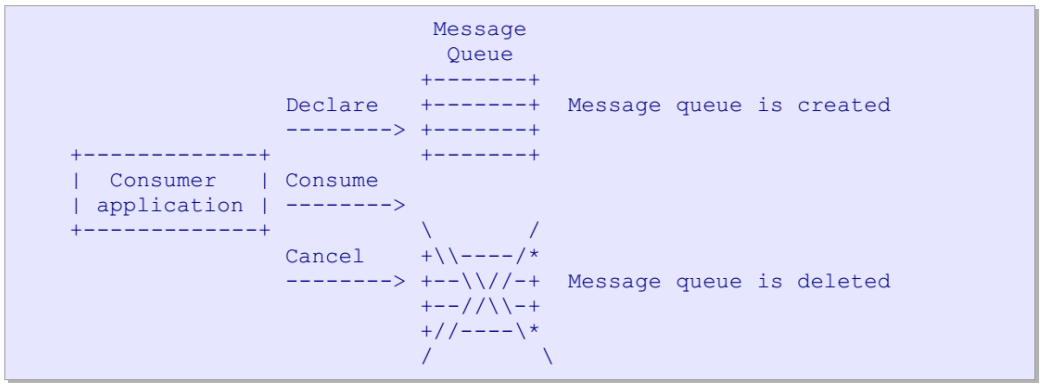
- name – 如果没有指定，服务器会选择一个名称，并将其提供给客户端。一般来说，当应用程序共享消息队列时，它们事先就消息队列名称达成一致，当应用程序处于自身目的需要一个消息队列时，它可让服务器提供一个名称。
- exclusive – 如果设置为 true，队列将只属于当前连接，并在连接关闭时删除。
- durable – 如果设置为 true，当服务器重启时，消息队列仍然保持为活动状态。如果服务器重启，它可能会丢失瞬时消息。

2.1.4.2 队列生命周期

这里主要有两种消息队列生命周期：

- 持久化消息队列：它们可被多个消费者共享，并可独立地存在。无论是否有消费者接收消息，消息队列都可以继续存在并收集消息。
- 临时消息队列：对某个消费者时私有的，并只能绑定那个消费者。当消费者断开连接时，消息队列会被删除。

这还有一些变化，如共享消息队列在最后一个消费者断开连接时会删除消息队列。下图展示了临时消息队列创建和删除的过程：



2.1.5 绑定

绑定是交换和消息队列之间的关系，它告诉交换器如何路由消息。绑定是由客户端应用程序命令（拥有和使用消息队列的应用程序）来绑到交换器上的。我们可以在伪代码中表达绑定命令，如下所示：

```
Queue.Bind <queue> TO <exchange> WHERE <condition>
```

让我们看下三种典型使用场景：共享队列、私有应答队列和发布订阅。

2.1.5.1 构建共享队列

共享队列是典型的中间件“点到点队列”。在 AMQP 中，我们可以使用默认交换器和默认绑定。让我们假设消息队列被称为“app.svc01”。这里是创建共享队列的伪代码：

```
Queue.Declare
    queue=app.svc01
```

在这个共享队列中，我们可能有很多消费者。为了从共享队列中消费，每个消费者可以这么做：

```
Basic.Consume
    queue=app.svc01
```

为了发送到共享队列，每个生产者将消息发送到默认的交换器：

```
Basic.Publish
    routing-key=app.svc01
```

2.1.5.2 构建应答队列

应答队列通常是临时的，服务器分配名称的。它们通常也是私有的，即只能由单个消费者读取。除了这些特殊情况外，应答队列使用与标准队列相同的匹配准则，因此我们也可以使用默认交换器。

下面是创建应答队列的伪代码，这里 S 表示一个服务器应答：

```
Queue.Declare
    queue=<empty>
    exclusive=TRUE
S:Queue.Declare-Ok
    queue=tmp.1
```

为了发布到应答队列，生产者将消息发送到默认交换器中：

```
Basic.Publish
    exchange=<empty>
    routing-key=tmp.1
```

有一个标准的消息属性“Reply-To”，它专门设计出来用于携带应答队列的名称。

2.1.5.3 构建 Pub-Sub 订阅队列

在经典的中间件中，术语“订阅（substrciption）”在概念上是模糊的，它至少涉及两个不同的概念：匹配消息的标准集和保存匹配消息的临时队列。AMQP 把这个工作分成了绑定和消息队列。在 AMQP 中没有称为“订阅”的实体。

让我们来描述 pub-sub 订阅：

- 为单个消费者（或某些情况下，多个消费者）保存消息
- 以不同的方式，通过一系列的匹配主题的绑定规则、消息字段或内容，从多个源中收集消息

订阅队列和命名或应答队列之间的关键区别是订阅队列名称与路由目的无关，并且路由是通过匹配准则来完成的，而不是一对一路由键值字段匹配来完成的。

我们采用常见的主题树 pub-sub 模型并对其实现。我们需要一个能够在主题树上匹配的交换器类型。在 AMQP 中，这就是“topic”交换器类型。“topic”交换器会匹配类似“STOCK.USD.*”通配符，就像“STOCK.USD.NYSE”这样的路由键。

我们不能使用默认的交换器或绑定，因为它们不会做“topic”风格的路由。因此我们必须明确创建一个绑定。下面是创建和绑定 pub-sub 订阅队列的伪代码：

```
Queue.Declare
    queue=<empty>
    exclusive=TRUE
S:Queue.Declare-Ok
    queue=tmp.2
Queue.Bind
    queue=tmp.2
    TO exchange=amq.topic
    WHERE routing-key=STOCK.USD.*
```

为了从订阅队列中消费，消费者需要这样做：

```
Basic.Consume
    queue=tmp.2
```

当发布一个消息时，生产者需要像这样做：

```
Basic.Publish
    exchange=amq.topic
    routing-key=STOCK.USD.ACME
```

Topic 交换器会使用其绑定表来处理传入的路由键（“STOCK.USD.ACME”），并找到一个匹配项“tmp.2”。然后它会将消息路由到那个订阅队列中。

2.2AMQP 命令架构

本章节解释了应用程序如何与服务端对话。

2.2.1 协议命令（类和方法）

中间件是复杂的，我们在设计协议架构的挑战是抑制复杂性。我们的方法必须基于类来对传统 API 进行建模，这个类中包含方法，并定义了方法明确应该做什么，怎么把它做好。这就导致产生了大量的命令集，但是每个命令都应该比较容易理解。

AMQP 命令被组进类中。每个类覆盖了一个特定功能域。有些类是可选的 – 每个节点都实现了需要支持的类。

这有两个不同的方法对话：

- 同步请求-响应，在其中一个节点发送请求，另一个节点发送应答。同步请求和响应方法适用于性能不是关键的地方。
- 异步通知，一个节点发送方法但不期望得到应答。异步方法适用于性能是至关重要的地方。

为了使方法处理简单，我们为每个异步请求定义了不同的应答。也就是说，没有方法适用于两个不同请求的应答。这就意味着一个节点发送同步请求后，可以接收和处理传入的方法，直到得到一个有效的同步应答。这使得 AMQP 与更多传统的 RPC 协议是由区别的。

方法可以形式上定义为同步请求、同步应答（针对特定请求）或者异步的。最后，每个方法形式上被定义为客户端（服务器到客户端），或服务端（客户端到服务器）。

2.2.2 映射 AMQP 到中间件 API

我们已经设计了 AMQP 是可映射到中间件的 API。这种映射有一些事智能的（不是所有方法，也不是所有参数对应用程序都是有意义的），也有一些事机械的（给定一些规则，所有方法可以在无人工干预的情况下映射）。

这么做的优势是对于那些已经了解 AMQP 语义（本章描述的类）的开发者会在他们使用的环境中找到相同的语义。

例如，下面是 Queue.Declare 方法示例：

```
Queue.Declare
    queue=my.queue
    auto-delete=TRUE
    exclusive=FALSE
```

这能转换成线级帧：

```
    +--------+---------+----------+-----------+-----------+
    | Queue  | Declare | my.queue |     1     |     0     |
    +--------+---------+----------+-----------+-----------+
      class     method     name    auto-delete  exclusive
```

或者更高级别的 API：

```
queue_declare (session, "my.queue", TRUE, FALSE);
```

映射为异步方法的伪代码逻辑：

```
send method to server
```

映射为同步方法的伪代码逻辑：

```
send request method to server
repeat
    wait for response from server
    if response is an asynchronous method
        process method (usually, delivered or returned content)
    else
        assert that method is a valid response for request
        exit repeat
    end-if
end-repeat
```

值得一提的是，对于大多数应用程序，中间件可以完全隐藏在技术层面中，并且实际 API 使用的影响会小于中间件的健壮性和能力性。

2.2.3 无确认

聊天式协议（chatty protocol，是一个应用程序或路由协议，需要客户端或服务器在得到一个确认之后才可以再次发送）是慢的。在那些性能是问题的场景下，我们会使用大量地使用异步方式。通常，我们从一个节点发送内容到另外的节点。我们会尽可能快地发出方法，而不等待确认。在必要时，我们在更高层次上实现窗口和流控，比如在消费者级别。

我们可以免除确认，应为我们为所有行为采用了断言模型。要么它们成功，要不就是我们在关闭通道和连接中有异常。

AMQP 中可以没有确认。成功是寂静的，而失败是喧闹的。当应用程序明确需要追踪成功和失败时，它们应该使用事务。

2.2.4 连接类

AMQP 是一个有连接协议。连接被设计为持久的，并且可以运载多个通道。连接生命周期是这样的：

- 客户端打开到服务端的 TCP/IP 连接并发送一个协议头。这只是客户端发送的数据，而不是作为方法格式的数据。
- 服务端使用自身协议版本和其他属性，包括它支持的安全机制列表（Start 方法）进行响应。
- 客户端选择一种安全机制（Start-Ok）。
- 服务端开始认证过程，它使用 SASL 的质询-响应模型（challenge-response model）。服务端向客户端发送一个质询（Secure）。
- 客户端向服务端发送一个认证响应（Secure-Ok）。例如，对于使用“plain”机制，影响包含登录用户和密码。
- 服务端重复质询（Secure）或转到协商，发送一系列参数，如最大帧大小（Tune）。
- 客户端接收或降低这些参数（Tune-Ok）。
- 客户端正式打开连接并选择一个虚拟主机（Open）。
- 服务端确认虚拟主机是一个有效选择（Open-Ok）。
- 客户端现在按需使用连接。
- 一个节点（客户端或服务端）结束连接（Close）。
- 另一个节点握手连接结束（Close-Ok）。
- 服务端和客户端关闭它们各自的套接字连接。

这里没有为不完全打开的连接上的错误进行握手。根据成功协议头协商（后续有详细定义），在发送或接收到 Open 或 Open-Ok 之前，如果一个节点检测到错误，这个节点必须关闭 socket，并不发送任何后续的数据。

2.2.5 通道类

AMQP 是一个多通道协议。通道提供了一种将重量级 TCP/IP 连接复用为几个轻连接的方式。这使得协议对于防火墙更加友好，因为端口使用时可预测的。这同样意味着传输调整和网络服务质量可以得到更好的利用。

通道是独立的，它们可以同时执行不同的功能，可用带宽会在当前活动之间共享。

这是令人期待和鼓舞的，多线程客户端应用程序经常使用“每个线程一个通道”编程模型。然而，从单个客户端打开许多连接到一个或多个 AMQP 服务端是完全可以接受的。通道生命周期如下：

1. 客户端打开新通道（Open）。
2. 服务端确认新通道准备就绪（Open-Ok）。
3. 客户端和服务端可以按需使用通道。
4. 一个节点（客户端或服务端）关闭通道（Close）。
5. 另一个节点对通道关闭进行握手（Close-Ok）。

2.2.6 交换类

交换器类让应用程序管理服务器上的交换器。这个类让应用程序脚本自己的连接（而不是依赖一些配置借口）。注：大多数应用程序不需要这个级别的复杂度，传统的中间件是不太可能能够支持这种语义。

下面是交换器的生命周期：

1. 客户端请求服务端确保交换器是否存在（Declare）。客户端可以细化到，“如果交换器不存在则创建”，或“如果交换器不存在警告我，不需要创建”。
2. 客户端发布消息到交换器。
3. 客户端可选择删除交换器（Delete）。

2.2.7 队列类

队列类让应用程序管理服务器上的消息队列。这几乎是所有消费消息的应用程序中的基本步骤，至少要验证期望消息队列是否实际存在。

持久化消息队列的生命周期相当简单，如下：

1. 客户端断言消息队列存在（Declare，使用“passive”参数）。
2. 服务端确认消息队列存在（Declare-Ok）。
3. 客户端从消息队列中读取消息。

临时消息队列的生命周期更加有趣，如下：

1. 客户端创建消息队列（Declare，不提供队列名称，服务器会分配一个名称）。服务端确认（Declare-Ok）。
2. 客户端在消息队列上开启一个消费者。消费者的精确功能是由基础类定义的。

3. 客户端取消消费者，要么显示取消，要不通过关闭通道或链接隐式取消。
4. 当最后一个消费者从消息队列中消失，同时超过礼貌性超时时，服务端会删除消息队列。

AMQP 像消息队列一样为主题订阅实现分发机制。这使得结构更加有趣，订阅可以在合作订阅应用程序池中进行负载均衡。

订阅生命周期会涉及到额外的绑定阶段：

1. 客户端创建消息队列（Declare），服务端进行确认（Declare-Ok）。
2. 客户端绑定消息队列到 topic 交换器（Bind），服务端进行确认（Bind-Ok）。
3. 客户端像前面的例子来使用消息队列。

2.2.8 基础类

基础类实现本规范中描述的消息功能。它支持如下主要语义：

- 从客户端发送消息给服务端，异步发生（Publish）。
- 启动和停止消费者（Consume，Cancel）。
- 从服务端发送消息给客户端，异步发生（Deliver，Return）。
- 应答消息（Ack，Reject）。
- 同步从消息队列中取消息（Get）。

2.2.9 事务类

AMQP 支持两种类型的事务：

1. 自动事务，每个发布的消息和应答都处理为独立事务。
2. 服务器本地事务，服务端会缓存发布的消息和应答，并且根据需要由客户端来提交它们。

事务类（“tx”）使应用程序可访问第二种类型，即服务器事务。这个类的语义如下：

1. 应用程序要求在每个通道中都有事务（Select）。
2. 应用程序做一些工作（Publish，Ack）。
3. 应用程序提交或回滚工作（Commit，Roll-back）。
4. 应用程序做一些工作，循环往复。

事务能覆盖发布内容和应答，但不能覆盖投递。因此，回滚不能将任何消息重新入队或者重新投递，客户端有权在事务中确认这些信息。

2.3AMQP 传输架构

本章节解释了命令是如何映射到线级协议的。

2.3.1 总体描述

AMQP 是二进制协议。信息被组织成各种类型的帧（frames）。帧可以携带协议方法和其他信息。所有帧都有同样的格式：帧头（frame header）、帧负载（frame payload）和帧尾（frame end）。帧负载的格式依赖于帧类型。

我们假设有一个可靠的面向流的网络传输层（TCP/IP 或等价的）。

在单个套接字连接中，那里可以存在多个独立的控制线程，它们被称为通道。每帧都使用通道号来编号。通过交错它们的帧，不同的信道共享连接。对于任何给定的信道，帧以严格的顺序运行，可用于驱动一个协议解析器（通常是状态机）。

我们使用一组小的数据类型，如比特、整数、字符串和字段表来构造帧。帧字段被紧密包装，不会使得他们解析缓慢或复杂。从协议规范中机械地生成帧层是相对简单的。

线级格式被设计成可伸缩的和通用的，可以适用于任意高级协议（不仅仅 AMQP）。我们假设 AMQP 将扩展、改进，随时间的推移线级格式仍然会得到支持。

2.3.2 数据类型

AMQP 数据类型用于方法帧中，它们是：

- 整型（Integers，1 到 8 个字节），用来表示大小、数量、范围等等。整型通常是无符号的，在帧中可能是未对齐的。
- 位（Bits），用来表示开/关值。位被包装成字节。
- 短字符串（Short strings），用来保存短的文本属性。短字符串限制为 255 字节，可以在无缓冲区溢出的情况下进行解析。
- 长字符串（Long strings），用来保存二进制数据块。
- 字段表（Field tables），用来保存名称-值对（name-value pairs）。字段值可以是字符串、整型等等。

2.3.3 协议协商

AMQP 客户端和服务端可对协议进行协商。这意味着当客户端连接时，服务端可以提出某些选择，客户端可以接受或修改。当两个节点达成一致时，连接会继续进行。协商式一种有用的技术，因为它让我们可以断言和预调整。

在 AMQP 中，我们就若干协议的具体方面进行协商：

- 真实协议和版本。服务端可能在同一个端口上托管多个协议。
- 加密参数和双方的认证。这是功能层的一部分，以前解释过。
- 最大帧大小，通道数量以及其他操作限制。

约定的限制可以使双方预先分配关键缓冲区，以避免死锁。每个传入的帧要么遵守约定的限制（这是“安全”的），要么超过“限制”，在这种情况下，另一方是错误的，必须断开连接。这是非常符合“它要么正常工作，要么完全不工作“的 AMQP 哲学。

两个节点达成一致的最先限度，如下：

- 服务端必须告诉客户端它提出了什么限制。
- 客户端进行响应，并可能减小其连接的限制。

2.3.4 限制帧

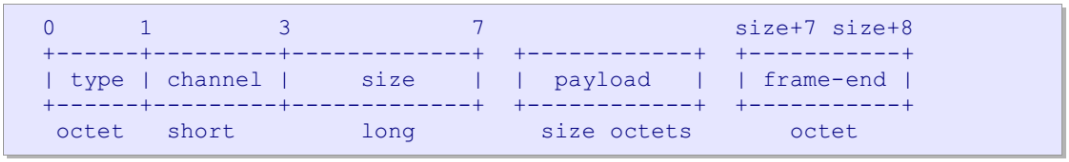
TCP/IP 是一个流协议，即没有限制帧的内建机制。现有协议可以在多个不同方式下解决这个问题：

- 每个连接中发送单个帧。这很简单，但很慢。

- 添加帧限定符到流中。这很简单，但解析很慢。
- 计算帧大小，并在每个帧的前面发送大小。这很简单和快速，是我们的选择。

2.3.5 帧细节

所有帧都由一个头（header，7 个字节），任意大小的负载（payload）和一个检测错误的帧尾字节组成：



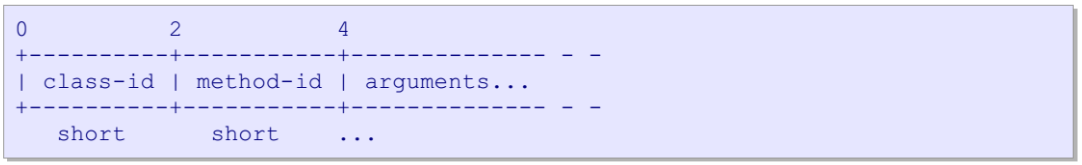
要读取一个帧，我们必须：

1. 读取帧头，并且检查帧类型和通道。
2. 根据帧类型，我们读取帧负载并进行处理。
3. 读取帧尾字节。

在实际实现中，当性能是一个关注点时，我们将使用“预读缓冲”或“收集读取”来避免为了读取一个帧而做三个独立的系统调用。

2.3.5.1 方法帧

方法帧可以携带高级协议命令（我们称之为方法）。一个方法帧携带一个命令。方法帧负载有下面的格式：



要处理一个方法帧，我们必须：

1. 读取方法帧负载。
2. 将其拆包成结构。给定的方法通常有相同的结构，因此我们可以快速地对方法进行拆包。
3. 检查方法在当前上下文中是否允许出现。
4. 检查方法参数是否有效。
5. 执行方法。

方法帧体（frame bodies）由 AMQP 数据字段（位、整型、字符串和字符串表）构成。编组代码直接从协议规范生成，可以非常迅速的。

2.3.5.2 控制帧

内容是我们通过 AMQP 服务器在端到端之间携带的应用程序数据。粗略地说，内容是由一组属性加上一个二进制数据部分组成。它所允许的属性集合由基本类定义，而这些属性

的形式“内容头帧”。其数据可以是任何大小，可可能被分解成几个（或多个）块，每一个都形成了内容体帧。

看一个特性通道的帧，当他们在线路上传输时，我们可能会看到像下面这样的东西：

```
[method]
[method] [header] [body] [body
[method]
...
```

某些方法（如 Basic.Publish、Basic.Deliver 等等）通常情况下被正式地定义为承载内容。当一个节点发送像方法帧这样的数据时，它总是会遵循一个内容头帧和零个或多个内容体帧这样的形式。

一个内容头帧有下面的格式：

```
0          2        4           12                14
+----------+--------+-----------+----------------+------------ - -
| class-id | weight | body size | property flags | property list...
+----------+--------+-----------+----------------+------------ - -
   short     short    long long       short        remainder...
```

我们将内容体放置在不同的帧中（并不包含在方法中），因此 AMQP 可以支持零拷贝技术，这样其内容就不需要编组或编码。我们将内容属性放置在它们自己的帧里，以便收件人可以选择性地丢弃不希望处理的内容。

2.3.5.3 心跳帧

心跳是一种设计用来撤销 TCP/IP 特性的技术，也就是说，在经过长时间的超时后，它有能力通过关闭代理物理连接来进行恢复。在某些情况下，我们需要快速直到节点是否断开连接，或者处于其他原因不能响应了。因为心跳可以在较低层次上进行，我们在传输层次上按节点交换的特定帧来实现，而不是按类方法。

2.3.6 错误处理

AMQP 使用异常来处理错误。任何操作错误（未找到消息队列、访问权限不足等）都会导致通道异常。任何结构化错误（非法参数、坏序列方法等）都会导致连接异常。异常会关闭通道或连接，同时也会返回响应码和响应文本给客户端应用程序。我们使用类似于 HTTP 和其他大多数协议中的三位回复代码加文字答复文本方案。

2.3.7 关闭通道和连接

连接或者通道，对于客户端来说，当它发送 Open 协议包时则被认为是打开的，对于服务端来说，当它发送 Open-Ok 协议包时则被认为是打开的。基于这一点，一个希望关闭通道或连接的对等方必须使用握手协议来这么做。

出于任何原因，可能会正常地或异常地冠以连接或通道，此操作必须小心。对于突然关闭并不总是很快被检测出来，并且在发生异常之后，我们可能会丢失错误应答代码。正确的设计是对于所有关闭必须进行握手，因此只有在我们确信对方了解情况之后，才关闭连接或

通道。

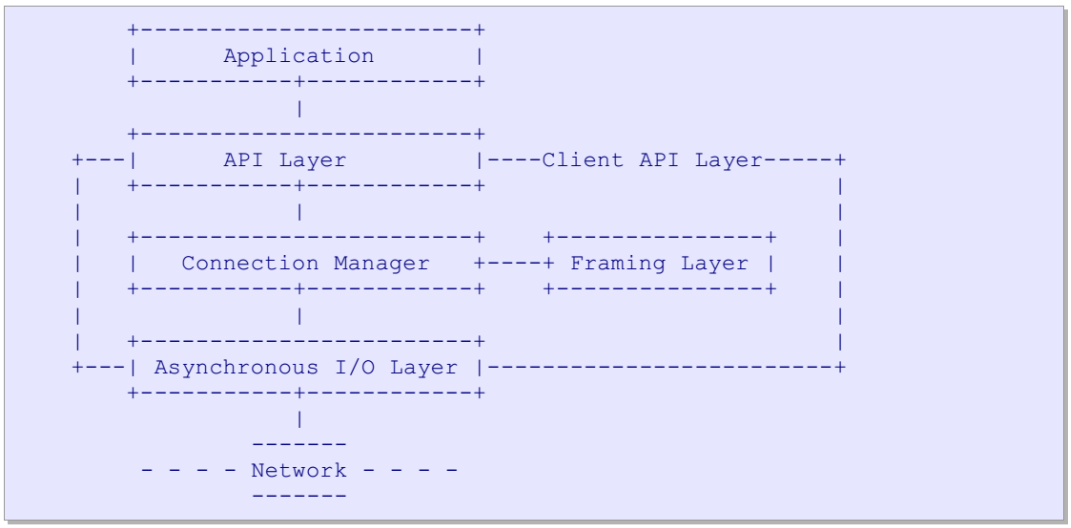
2.4AMQP 客户端架构

从应用程序中直接读写 AMQP 帧是可能的，但这是相当糟糕的设计。

即使是最简单的 AMQP 对话也比较复杂（同 HTTP 比较），应用程序开发者不应该为了向消息队列发送消息，而来理解二进制帧格式。推荐的 AMQP 客户端架构由以下几个抽象层组成：

1. 帧层（frame layer），此层采用 AMQP 协议方法，并按某种特定语言的格式（结构、类等等），来序列化成线级帧。帧层可以根据 AMQP 规范中机械式的生成（这是专为 AMQP 设计用 XML 实现的一个协议建模语言）。
2. 连接管理层（connection manager layer），此层用于读写 AMQP 帧，管理所有连接和会话逻辑。在此层中，我们可以封装打开连接和会话、错误处理、内容传输和接收的全逻辑。此层的大部分都可通过 AMQP 规范来自动化生成。例如，规范定义了那些方法可以携带内容，因此像发送方法和可选的发送内容这样的逻辑可以机械的生成。
3. API 层（API layer），此层暴露了应用程序工作的具体 API。API 层可能反应一些现有的标准，或者可能暴露高级 AMQP 方法，对本节前面介绍的内容做一个映射。AMQP 方法设计为使这些映射更加简单有用。API 层本身可能由多个层组成，如构建于 AMQP 方法之上的高级 API。

此外，通常还会有一些 I/O 层，这层可以是非常简单（同步套接字读取和写入）或复杂（全异步多线程 I/O）。下图显示了整体推荐的架构：



在本文中，当我们说客户端 API 时，我们指的则是应用程序下的所有层（I/O 层、帧层、连接管理层和 API 层）。我们通常将客户端 API 和应用程序分开说，应用程序会使用客户端 API 来与服务端进行对话。

3 功能说明

3.1 服务器功能说明

3.1.1 消息和内容

消息是中间件路由和队列系统处理的原子单元。消息携带一个内容，它由一个内容头、一组属性、一个内容体和一个持有不透明的二进制数据块所组成。

一个消息可以对应于许多不同的应用程序实体：

- 一个应用程序级别消息
- 一个传输文件
- 一个数据流帧等

消息可以持久化。持久化的消息安全地存储在磁盘中，并且即使存在严重的网络故障，服务器崩溃、泄露等情况下可以确保被投递。

消息可以具有优先级。高优先级消息在等待同一个消息队列时先于低优先级消息被发送出去。为了确保特定的服务质量级别，必须丢弃消息时，服务器将首先丢弃低优先级消息。

服务器不能修改接收到并传递给消费者应用程序的消息内容体。服务器可能增加内容头信息，但是不能删除或修改已经存在的信息。

3.1.2 虚拟主机

虚拟主机是服务器内的数据分区，它为期望在共享基础设施上提供 AMQP 服务的管理带来了便利。

虚拟主机包括自己的名称空间、一组交换器、消息队列和所有相关对象。每个连接必须关联单个虚拟主机。

在认证后，客户端可在“Connection.Open”方法中选择虚拟主机。这意味着，服务端的认证方案在此服务器上的所有虚拟主机上共享。然而，对于每个虚拟主机，使用的认证方案可能是唯一的。这将有助于共享主机基础设施。对于每个虚拟主机需要不同的认证方案的管理员应该使用单独的服务器。

连接中的所有通道在同一个虚拟主机上工作。同一个连接中，没有办法与不同的虚拟主机进行通信，也没有任何办法在不断开连接重新开始的情况下切换到其他虚拟主机。

协议没有提供用于创建或配置虚拟主机的机制-这是在服务器中以未定义的方式完成的，完全依赖于实现。

3.1.3 交换器

交换器是虚拟主机内的消息路由代理。交换器实例（我们通常称之为“交换器”）接收消息和路由信息 – 主要是路由键 – 并且要么投递消息到消息队列中，要么投递消息到内部服务。交换器是基于每个虚拟主机命名的。

应用程序可以在它们权限范围内自由地创建、共享、使用和销毁交换器实例。

交换器可能是持久的、临时的或者自动删除的。持久化的交换器会持续到它们被删除时。临时交换器会持续到服务器关闭时。自动删除交换器会持续到它们不再使用时。

服务器提供了一组特定的交换器类型。每个交换器类型实现了如下节中定义的特定匹配

和算法。AMQP 授权了少量的交换器类型，并推荐了一些。此外，每个服务器实现可能添加自己的交换器类型。

交换器能并发地路由单个消息到很多消息队列中。这将创建多个被独立消费的消息实例。

3.1.3.1Direct 交换器类型

Direct 交换器按如下方式来工作：

1. 消息队列使用路由键（K）绑定到交换器
2. 发布者使用路由键（R）发送消息到交换器
3. 如果 K=R 时，消息被投递到消息队列中

服务器必须实现 direct 交换器类型，并且在每个虚拟主机中必须预定义至少两个 direct 交换器：一个命名为“amq.direct”，另一个没有公共名称（为 Publish 方法作为默认交换器提供服务）。

注意：消息队列可以使用任何合法路由键值进行绑定，但是经常消息队列使用它们自己的名称作为路由键进行绑定。

特别地，所有消息队列必须使用消息队列的名称作为路由键自动绑定到匿名交换器上。

3.1.3.2Fanout 交换器类型

Fonout 交换器按如下方式来工作：

1. 消息队列不适用参数来绑定到交换器
2. 发布者发送消息到交换器
3. 消息被无条件地投递到消息队列中

Fanout 交换器微不足道的设计和实现。此交换器类型和预声明的交换器称为“amp.fanout”，它时强制的。

3.1.3.3Topic 交换器类型

Topic 交换器按如下方式来工作：

1. 消息队列使用路由模式（P）绑定到交换器
2. 发布者使用路由键（R）发送消息到交换器
3. 如果 R=P 时，消息被投递到消息队列中

用于 topic 交换器的路由键必须由零个或者多个点号分隔的单词组成。每一个单词可能包含字母 A-Z 和 a-z 和数字 0-9。

路由模式与路由键遵循相同的规则，*用于匹配单个单词，#用于匹配零个或多个单词。因此路由模式*.stock.#会匹配路由键 usd.stock 和 eur.stock.db，但不会匹配 stock.nasdaq。

对于 topic 交换器的设计，我们建议是保持所有已知路由键，当发布者使用新的路由键时，才更新此集合。对于给定的路由键，确定所有绑定是可能的，因此可为消息快速地找到消息队列。此交换器是可选的。

服务端应该实现 topic 交换器类型，在这种情况下，服务端必须在每个虚拟主机下预声明至少一个 topic 交换器，命名为 amq.topic。

3.1.3.4Headers 交换器类型

Headers 交换器按如下方式工作：

1. 消息队列使用包含匹配绑定的消息头和可选值的参数表来绑定交换器。在这种交换器中，不使用路由键。
2. 发布者发送消息到交换器，消息头属性包含名称和值的表。
3. 如果消息头属性与队列绑定的参数相匹配，消息被投递到队列中。

匹配算法是由参数表中的名称值对这样的特殊绑定参数来控制的。这个参数的名称是“x-match”。它可以采用两个值，以表示参数表中其他名称值对如何来进行匹配：

- “all”表示所有其他名称值对必须匹配与路由消息的头属性相匹配（即 AND 匹配）
- “any”表示只要消息头属性中的任何字段与参数表中的字段相匹配，消息应该被路由（即 OR 匹配）

绑定参数中的字段与消息中的字段相匹配，这样的情况包括：要么绑定参数中的字段没有值且消息头中存在相同名称的字段，要么绑定参数的字段有值，消息头中存在相同名称的字段且具有相同的值。

任何以“x-”而不是“x-match”开头的字段为将来保留并会被忽略。

服务端应该事先 headers 交换器类型，在这种情况下，服务端必须在每个虚拟主机中预声明至少一个 headers 交换器，命名为“amq.match”。

3.1.3.5 系统交换器类型

系统交换器按如下方式工作：

1. 发布者使用路由键（S）发送消息到交换器中
2. 系统交换器将消息投递给系统服务（S）

系统服务以“amq.”开头，为 AMQP 保留使用。所有其他名称都可以由服务器实现自由使用。此交换器类型是可选的。

3.1.3.6 实现定义的交换器类型

所有非规范交换器类型必须以“x-”开头。不以“x-”开头的交换器类型作为将来 AMQP 标准保留使用。

3.1.4 消息队列

消息队列是一个名为 FIFO 的缓冲区，它为一组消费者应用程序保存消息。在应用程序权限范围内，应用程序可以自由地创建、共享、使用和销毁消息队列。注意，在一个队列中存在多个读者的情况，或存在客户端事务 ，或使用优先级字段，或使用消息选择器，或特定实现投递优化（队列可能不会展示出 FIFO 特性）。唯一确保 FIFO 的方式是只有一个消费者连上队列。在那些情况下，队列可能被描述为“弱-FIFO”。

消息队列可能是持久化的、临时的或自动删除的。持久化消息队列会持续到它们被删除为止。临时消息队列会持续到服务端关闭为止。自动删除消息队列会持续到它们不再使用为

止。

消息队列存储它们的消息在内存中、磁盘中或其他上述的组合中。消息队列基于每个虚拟主机来命名的。

消息队列保存消息，并且在一个或多个消费者客户端之间进行分发。路由到消息队列的消息从不发送到一个以上的客户端，除非在失败或拒绝后进行重发。

单个消息队列能在同一时间独立地保存不同类型的内容。也就是说，如果基础和文件内容发给了同一个消息队列，这将会作为请求独立地投递给消费端应用程序。

3.1.5 绑定

绑定是消息队列和交换器之间的关系。绑定特有的路由参数将告诉交换器哪些队列应该得到消息。当需要驱动消息流到它们的消息队列中，应用程序可创建和销毁绑定关系。绑定的寿命取决于它们定义的消息队列—当消息队列被销毁时，它的绑定也被销毁。Queue.Bind 方法的特定语义将依赖于交换器类型。

3.1.6 消费者

我们使用术语“消费者”来表示客户端应用程序和控制具体客户端应用程序如何从消息队列中接收消息的实体。当客户端启动一个消费者时，它就在服务端中创建出消费者实体。当客户端取消一消费者时，它就在服务端中销毁消费者实体。消费者属于单个客户端通道，并且使消息队列异步地发送消息给客户机。

3.1.7 服务质量

服务质量控制了消息如何快速地发送。服务质量依赖于被分发的内容类型。一般而言，在客户端应答消息前，服务质量使用预取的概念来指出多少消息或多少字节的数据将被发送。目标是提前发送消息数据，以减少延迟。

3.1.8 确认/应答

应答是一个从客户端应用程序到消息队列的正式信号，它表示消息已经被成功处理。这有两种可能的应答模型：

1. 自动的（Automatic），服务端在消息投递到应用程序后立即从消息队列中删除消息（通过 Deliver 或 Get-Ok 方法）。
2. 显式的（Explicit），客户端应用程序必须为已处理的每一个消息或批量消息发送应答方法。

客户端层可以以不同方式来实现显式应答，如只要收到了消息或应用程序表示消息已经被处理了。这些区别都不会影响 AMQP 或者互操作性。

3.1.9 流控

流控是一种用来阻止从节点过来的消息流的紧急过程。它在客户端和服务端之间都按同样的方式工作，并且通过 Channel.Flow 命令实现。流控是可以停止过度生产的发布者的机制。如果它使用消息应答（通常意味着使用事务），消费者可以使用更为优雅的预取窗口机制。

3.1.10 命名约定

这些约定规范了 AMQP 的实体命名。服务端和客户端必须遵守这些约定：

- 用户定义的交换器类型前缀必须是“x-”
- 标准交换器实例前缀是“amq.”
- 标准系统服务前缀是“amq.”
- 标准消息队列前缀是“amq.”
- 所有其他交换器、系统服务和消息队列名称都在应用程序空间中

3.2AMQP 命令说明（类和方法）

3.2.1 注释说明

AMQP 方法为了互操作性原因可能定义特定的最小值（如每个消息队列的消费者数量）。这些最小值被定义在每个类的描述中。

遵从 AMQP 的实现应该为这样的字段实现合理值，最小值只能用于最小能力的平台上。

语法使用这些标记：

- ‘S:’表明从服务端发送到客户端的数据或方法
- ‘C:’表明从客户端发送到服务端的数据或方法
- +term 或+(…)表达式表示一个或多个实例
- *term 或*(…)表达式表示零个或多个实例

我们将方法定义为：

- 同步请求（“syn request”）。发送节点应该等待特定的响应方法，但可以异步实现此方法；
- 同步响应（“syn reply for XYZ”）；
- 异步请求或响应（“async”）。

3.2.2 类和方法细节

这部分是由生成的文档“amqp-xml-spec.odt”所提供。

4 技术说明

4.1INNA 分配的端口号

IANA 为标准的 AMQP 的 TCP 和 UDP 分配了 5672 端口。UDP 端口保留用于未来组播实现中。

4.2AMQP 线级格式

4.2.1 正式协议语法

我们为 AMQP 提供了完整语法（这是 AMQP 提供的参考，跳到下一节，你会发现它更加有趣，那里详细说明不同的帧类型和格式）：

```
amqp                = protocol-header *amqp-unit

protocol-header     = literal-AMQP protocol-id protocol-version
literal-AMQP        = %d65.77.81.80             ; "AMQP"
protocol-id         = %d0                       ; Must be 0
protocol-version    = %d0.9.1                   ; 0-9-1

method              = method-frame [ content ]
method-frame        = %d1 frame-properties method-payload frame-end
frame-properties    = channel payload-size
channel             = short-uint                ; Non-zero
payload-size        = long-uint
method-payload      = class-id method-id *amqp-field
class-id            = %x00.01-%xFF.FF
method-id           = %x00.01-%xFF.FF
amqp-field          = BIT / OCTET
                    / short-uint / long-uint / long-long-uint
                    / short-string / long-string
                    / timestamp
                    / field-table
short-uint          = 2*OCTET
long-uint           = 4*OCTET
long-long-uint      = 8*OCTET
short-string        = OCTET *string-char        ; length + content
string-char         = %x01 .. %xFF
long-string         = long-uint *OCTET          ; length + content
timestamp           = long-long-uint            ; 64-bit POSIX
field-table         = long-uint *field-value-pair
field-value-pair    = field-name field-value
field-name          = short-string
field-value         = 't' boolean
                    / 'b' short-short-int
                    / 'B' short-short-uint
                    / 'U' short-int
                    / 'u' short-uint
                    / 'I' long-int
                    / 'i' long-uint
                    / 'L' long-long-int
                    / 'l' long-long-uint
                    / 'f' float
                    / 'd' double
                    / 'D' decimal-value
                    / 's' short-string
```

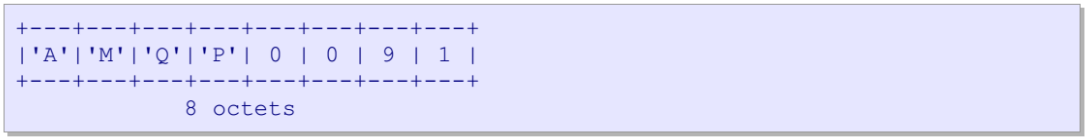
```
                    / 'S' long-string
                    / 'A' field-array
                    / 'T' timestamp
                    / 'F' field-table
                    / 'V'                       ; no field
boolean             = OCTET                     ; 0 = FALSE, else TRUE
short-short-int     = OCTET
short-short-uint    = OCTET
short-int           = 2*OCTET
long-int            = 4*OCTET
long-long-int       = 8*OCTET
float               = 4*OCTET                   ; IEEE-754
double              = 8*OCTET                   ; rfc1832 XDR double
decimal-value       = scale long-uint
scale               = OCTET                     ; number of decimal digits
field-array         = long-int *field-value     ; array of values
frame-end           = %xCE

content             = %d2 content-header *content-body
content-header      = frame-properties header-payload frame-end
header-payload      = content-class content-weight content-body-size
                      property-flags property-list
content-class       = OCTET
content-weight      = %x00
content-body-size   = long-long-uint
property-flags      = 15*BIT %b0 / 15*BIT %b1 property-flags
property-list       = *amqp-field
content-body        = %d3 frame-properties body-payload frame-end
body-payload        = *OCTET

heartbeat           = %d8 %d0 %d0 frame-end
```

我们使用了在 IETF RFC 2234 中定义的扩展 BNF 语法。归纳起来：

- 规则的名称仅仅是它名称本身。
- 终端由一个或多个数字字符指定的，这些字符的基本解释为“d”或“x”。
- 规则可以通过列举规则名称的序列来定义简单的、有序字符串的值。
- 使用破折号（“–”）来表明替换值的范围，可替换的范围可以简洁地指定。
- 在圆括号中的元素被当作单个元素，其内容是严格保序的。
- 由斜杠（“/”）分隔的元素是可替换的。
- 在元素之前的操作符“*”表示重复。完整格式为：“<a>*<b>element”，这里<a>和<b>是可选的十进制值，表示只能出现大于<a>而小于<b>的元素。
- 规则格式：“<n>element”等价于“<n>*<n>element”。
- 方括号中的元素是可选元素序列。

4.2.2 协议头

客户端必须通过发送协议头来开始新的连接。它是八字节序列：



协议头由大写字母“AMQP”，其后面跟着常量组成：

1. 协议主版本号，按照章节 1.4.2 中使用
2. 协议次版本号，按照章节 1.4.2 中使用
3. 协议修订版本号，按照章节 1.4.2 中使用

协议协商模型与现有的协议（如 HTTP 协议）兼容，它使用常量文本字符串来发起一个连接，并为了决定应用什么规则使用防火墙来发现协议的开始。

客户端和服务端通过以下方式来与协议和版本达成一致：

- 客户端打开到 AMQP 服务器的新套接字连接，并发送协议头
- 服务端可接收或拒绝协议头。如果它会写入合法的协议头并发送给套接字来拒绝协议头，然后关闭套接字。
- 否则，它会同意套接字打开，并相应地实现协议。

示例：

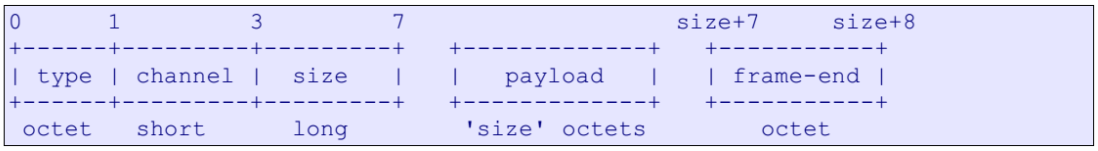
```
Client sends:            Server responds:
AMQP%d0.0.9.1            Connection.Start method
AMQP%d0.1.0.0            AMQP%d0.0.9.1<Close connection>
HTTP                     AMQP%d0.0.9.1<Close connection>
```

实现者的指导方针：

- 服务端可能接受非 AMQP 协议，如 HTTP。
- 如果服务端无法识别套接字的前五个字节，或者不支持客户端请求的特定协议版本，它必须写合法的协议头到套接字中，然后同步到套接字中（确保客户端应用程序收到数据），然后再关闭套接字连接。服务端可能打印诊断信息用来辅助调试。
- 客户端可能通过使用最高支持版本的连接尝试来检测服务端协议版本，并且如果收到从服务端发回这种较低版本的信息，客户端会使用较低版本进行重连。
- 实现多 AMQP 版本的客户端和服务端都应该使用全部八字节协议头来标识协议。

4.2.3 通用帧格式

所有帧都以七个字节的头开始，其中包括一个类型字段、一个通道字段和大小字段：



AMQP 定义如下帧类型：

- Type = 1，“METHOD”：方法帧
- Type = 2，“HEADER”：内容头帧
- Type = 3，“BODY”：内容体帧
- Type = 4，“HEARTBEAT”：心跳帧

通道号为零的代表全局连接中的所有帧，通道号 1-65535 代表特定通道的帧。

大小字段是负载（payload）的大小，不包括帧尾字节。由于 AMQP 假设是一个可靠的有连接协议，我们使用帧尾来检测由于错误客户端或服务端实现引起的错误。

实现者的指导方针：

- 帧尾字节必须是十六进制值%xCE。
- 如果一个节点收到了未定义类型的帧，它必须将其视为致命的协议错误，并关闭连接而不是进一步发送任何数据。
- 当一个节点读取一个帧时，它必须在尝试帧解码前检查帧尾是否合法。如果帧尾非法，它必须将其视为致命的协议错误，并关闭连接而不是进一步发送任何数据。它应该记录问题信息，因为这些问题表明在服务端或者客户端的帧代码实现的错误。
- 一个节点不能发送超过约定大小的帧。节点收到超过大小的帧时，必须发出带有响

应码为 501（帧错误）的连接异常信号。

- 对于那些相关连接类的所有心跳帧、方法头帧、方法体帧，通道号必须为零。节点收到非零通道号的上述帧，必须发出带有响应码为 503（非法命令）的连接异常信号。

4.2.4 方法负载

方法帧体由可变数据字段列表组成，称为参数。所有方法体都以类和方法的标识号开始：

```
0          2           4
+----------+-----------+-------------- - -
| class-id | method-id | arguments...
+----------+-----------+-------------- - -
   short      short     ...
```

实现者指导方针：

- Class-id 和 method-id 是由 AMQP 类和方法定义的常量。
- 参数是一组特定于每个方法的 AMQP 字段。
- Class-id 中%x00.01-%xEF.FF 范围的值被 AMQP 标准类保留使用。
- Class-id 中%xF0.01-%xFF.FF（%d61440-%d65535）范围的值可能用于非标准扩展类的实现。

4.2.5AMQP 数据字段

AMQP 由两种级别的数据字段规范：用于方法参数的原生数据字段和用于在字段表中应用程序之间的数据字段。字段表原生数据字段的的超集。

4.2.5.1 整型

AMQP 定义了以下原生整型类型：

- 无符号节点（8bits）。
- 无符号短整型（16bits）。
- 无符号长整型（32bits）。
- 无符号长长整型（64bits）。

整型和字符串长度从事无符号的，并且按网络字节序保存。当两个高低系统（如两个 Intel 处理器）相互对话时，我们不会对它们尝试优化。

实现者的指导方针：

- 实现者不能假设在帧内的整型编码在内存字节边界是对齐的。

4.2.5.2 位

AMQP 定义了原生位字段类型。位累积成整个字节。当帧内的两个或更多的位是连续时，它们将会包装成一个或多个字节，并且在每个字节的低位开始。这里没有要求帧内的所有位

是连续的，但这么做通常可以减小帧大小。

4.2.5.3 字符串

AMQP 字符串是可变长度的，由一个整数长度后跟零个或多个字节数据表示。AMQP 定义了两种原生字符串类型：

- 短字符串，以 8 位无称号整型长度后跟零个或多个字节数据存储. 短字符串可携带最多 255 字节的 UTF-8 数据, 但不能包含二进制零字节.
- 长字符串，以 32 位无称号整型长度后跟零个或多个字节数据存储. 长字符串可包含任意数据。

4.2.5.4 时间戳

时间戳使用精度为一秒的 64 位 POSIX 的 time_t 格式来保存。通过使用 64 位，我们可以避免未来因 31 位和 32 位 time_t 值相关引起的问题。

4.2.5.5 字段表

字段表是包含名称-值对的长字符串。名称-值对通过短字符串定义名称，字节定义值类型和值来编码。有效的表字段类型在语法上是原生整型、位、字符串和时间戳类型的扩展。多字节整型字段总是按网络字节序保存。

实现者的指导方针：

- 字段名称必须以字母开头，其后可能跟‘$’、‘#’、数字或下划线，最大长度为 128 个字符。
- 服务端应该验证字段名称，并在收到无效的字段名称时，它应该发出带有响应码为 503（语法命令）的连接异常信号。
- 十进制值并不支持浮点值，而是固定的业务值，如货币汇率和金额。它们按字节进行编码，代表了位置编号，其后跟着一个有符号的长整型。十进制字节是无符号的。
- 重复字段是非法的。对一个包含重复字段的节点行为是未定义的。

4.2.6 内容帧

某些特定的方法（Publish、Deliver 等）会携带内容。请参考“功能说明”章节对于每个方法的说明，并且它们是否是携带内容的方法。携带内容的方法无条件地这么做。

内容由一个或多个帧组成，如下：

1. 一个内容头帧提供内容属性。
2. 可选择性的有一个或多个内容体帧。

特定通道上的内容帧是严格有序的。也就是说，它们可能与其他通道的帧混合，但同一个通道的两个内容帧不可能混合或重叠的，也不可能出现单个内容的内容帧与相同通道上的方法帧混合。

注意：任何非内容帧都会明确地标识内容的结束。尽管从内容头中直到内容的大小，但

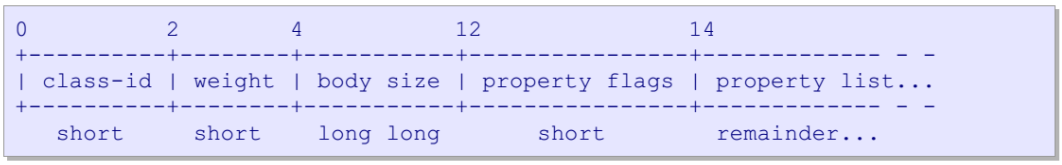
允许发送者在不需要关闭通道的情况下中止内容的发送。

实现者的指导方针：

- 节点收到不完整或错误格式的内容时，它应该抛出带有响应码为 503（非期望帧）的连接异常。这包括缺失的内容头、内容头中的错误类 ID 以及缺失的内容体帧等。

4.2.6.1 内容头

内容头负载有下面的格式：

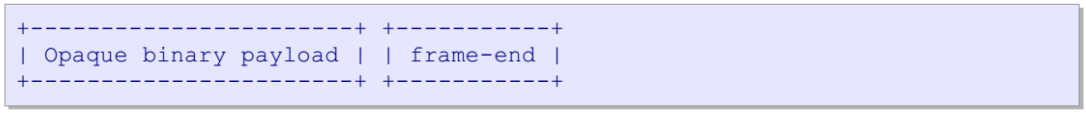


实现者的指导方针：

- Class-id 必须与方法帧的 class id 匹配。通过抛出带有响应码为 501（帧错误）的连接异常，节点必须响应一个无效的 class-id。
- Weight 字段是未使用的且必须为零。
- Body size 字段值是一个 64 位值，它定义了内容体的总大小，也就是说，后面内容体帧的体大小的总和。零表示无内容体帧。
- Property flags 字段是一个位数组，它表示在序列中的每个属性值的存在性。位是从最高到最低进行排序的，第 15 位代表第一个属性。
- Property flags 字段可指定多于 16 中属性。如果最后位被设置了，这表示其后有进一步的属性标志字段。这里有很多根据需要的属性标志字段。
- 属性值是 AMQP 特定类的数据字段。
- 位属性仅仅通过他们各自的属性标志（1 或 0）来表示，并且在属性列表中从不存在。
- 内容帧中的通道号不能为零。节点在收到内容帧内通道号为零时，必须发出带有响应码为 504（通道错误）的连接异常信号。

4.2.6.2 内容体

内容体负载是一个不透明的二进制块，其后跟着帧尾字节：



内容体可以根据需要分成多个帧。在连接协商时，帧负载的最大值大小是由两端达成一致的。

实现者的指导方针：

- 节点必须将内容体分成多个帧来处理，它们作为单个集合来存储这些帧，并且要么分成更小的帧进行传输，要么连接成单个块分发给应用程序。

4.2.7 心跳帧

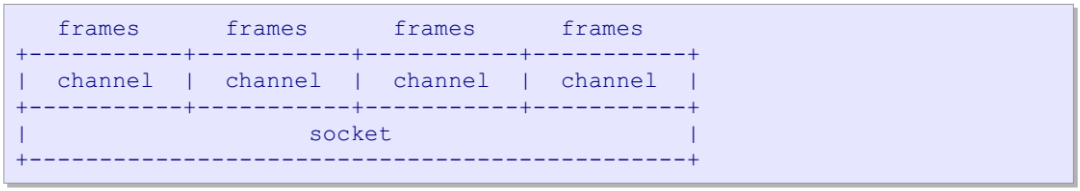
心跳帧告诉收件人发件人仍然是活的。在连接调整期间，心跳帧的速率和定时是可以协商的。

实现者的指导方针：

- 心跳帧必须有一个编号为零的通道号。节点在收到无效心跳帧时必须使用响应码 501（帧错误）抛出连接异常。
- 如果节点不支持心跳，它必须在不发出任何错误和故障的情况下丢弃心跳帧。
- 客户端应该在收到 Connection.Tune 方法后开始发送心跳，并且在收到 Connection.Open 后开始监控心跳。服务端应该在收到 Connection.Tune-Ok 后开始发送和监控心跳。
- 节点应该按指定间隔尽最大努力来发送心跳。心跳可以在任何时候被发送出去。任何发送字节都可以是心跳的有效替代，因此当超过心跳间隔还没有发送非 AMQP 心跳时，心跳必须被发送。如果节点在两个心跳间隔或更长时间内未检测到传入的报文，它应该在没有进行 Connection.Close/Close-Ok 握手下关闭连接，并且记录错误。
- 心跳应该持续到连接套接字关闭为止，包括在 Connection.Close/Close-Ok 握手期间和之后。

4.3 通道复用

AMQP 允许节点创建多个独立的控制线程。每个通道可作为虚拟连接来共享单个套接字：



实现者的指导方针：

- AMQP 节点可能支持多个通道。在连接协商期间，通道最大值数目可被定义，节点可能协商这个数目到一。
- 每个节点都应该在以公平的方式平衡所有打开通道的流量。这种平衡方式可以基于每帧为基础，或者基于每个通道上的流量为基础来做。节点不应该允许一个非常繁忙的通道来让不太繁忙的通道饿死。

4.4 可见性保证

服务端必须确保客户端对服务端状态的观察相一致：

下面的示例说明了在这种情况下客户端的观察意味着什么：

- 客户端 1 和客户端 2 连接到相同的虚拟主机
- 客户端 1 声明一个队列
- 客户端 1 收到 Declare.Ok（一个观察的例子）
- 客户端 1 告诉客户端 2 关于自己声明了队列

- 客户端 2 对同一个队列做了被动声明

可见性保证确保了客户端 2 能看到这个队列（在没有删除的情况下）。

4.5 通道关闭

当任何以下事情发生时，服务端会考虑关闭通道：

1. 任何一个节点使用 Close/Close-Ok 握手来关闭通道或父连接。
2. 任何一个节点在通道或父连接上抛弃异常。
3. 任何一个节点未使用 Close/Close-Ok 关闭父连接套接字。

当服务端关闭通道时，通道上任何没有应答的消息将标记为重新分发。

当服务端关闭连接时，它会删除那条连接所拥有的自动删除信息。

4.6 内容同步

在某些情况下，同步请求响应方法对相同通道上的异步内容投递产生影响，包括：

- Basic.Consume 和 Basic.Cancel 方法，会启动和停止消息队列中的消息流。
- Basic.Recover 方法，请求服务器重新投递消息到通道。
- Queue.Bind、Queue.Unbind 和 Queue.Purge 方法，会影响消息流进行消息队列。

实现者的指导方针：

- 请求-响应的影响在响应方法之前在通道上必须是不可见的，并且之后必须可见。

4.7 内容排序保证

流过通道的方法的顺序是稳定的：方法是按照它们他送时的顺序接收的。这是由 AMQP 使用的 TCP/IP 传输所保证的。此外，内容由服务器按稳定的方式来处理。尤其是，服务器内的经过单个路径的内容流保序。对于那些通过单个路径的给定优先级内容，我们定义内容处理路径，它由一个输入通道、一个交换器、一个队列和一个输出通道组成。

实现者的指导方针：

- 服务器必须保持内容流通过单个内容处理路径的顺序，除非在 Basic.Deliver 或 Basic.Get-Ok 方法上设置了 redelivered 字段，并且根据规则的条件下该字段可以被设置。

4.8 错误处理

4.8.1 异常

使用标准的异常编程模型，AMQP 不会发出成功信号，只在失败时才发出信号。AMQP 定义两种异常级别：

1. 通道异常，指那些关闭那些引起错误的通道。通道异常通常因为软错误引起的，这并不影响应用程序的其他部分。
2. 连接异常，指关闭那些通常因为硬错误引起的套接字连接，硬错误如程序错误、错

误配置或其他需要干预的情况。

我们在每个类和方法的定义中正式地记录了断言。

4.8.2 响应码格式

AMQP 响应码符合 IETF RFC 2821 中“响应码的严重程度和理论”的定义。

4.9 限制

AMQP 规范在将来的 AMQP 扩展或相同线级格式的协议做了以下限制：

- 每个连接上的通道数量：16 位的通道数量
- 协议类数量：16 位类 ID
- 每个协议类的方法数量：16 位方法 id

AMQP 规范对于数据做了以下限制：

- 短字符串的最大值大小：255 字节
- 长字符串或字段表的最大值大小：32 位大小
- 帧负载的最大值大小：32 位大小
- 内容的最大值大小：64 位大小

服务端或客户端也可以对资源做自己的限制，如并发连接数、每个通道的消费者数目、队列数量等等。这不影响互操作性，并且没有详细提出。

4.10 安全

4.10.1 目标和原则

我们通过在所有地方使用指定长度的缓冲区来防止缓冲区溢出攻击。任何数据读取时，所有外部提供的数据都可以根据允许的最大长度进行验证。可以通过关闭通道或连接来显式地处理无效数据。

4.10.2 拒绝服务攻击

AMQP 通过返回响应码然后关闭通道和连接来处理错误。这避免了错误出现后的模糊状态。在连接协商期间，服务器应该假设异常状态因为访问服务器的敌对尝试造成的。在连接协商期间，一般对任务异常状态的反应是暂停连接（可能是一个线程）几秒钟时间，然后关闭网络连接。这包括语法错误、数据过大和尝试认证失败。服务器应该记录所有这样的异常，并标记或阻止客户端引起多重错误。