**声明：此文档为个人翻译官方文档，未经允许请无外传。**

**翻译者：杨德军**

AMQP 0-9-1 Model Explained

**[About This Guide](http://www.rabbitmq.com/tutorials/amqp-concepts.html" \l "about-this-guide)**

本指南概述了AMQP 0-9-1协议，它是RabbitMQ支持的协议之一。

**[High-level Overview of AMQP 0-9-1 and the AMQP Model](http://www.rabbitmq.com/tutorials/amqp-concepts.html" \l "highlevel-overview)**

**[What is AMQP 0-9-1?](http://www.rabbitmq.com/tutorials/amqp-concepts.html" \l "what-is-amqp)**

AMQP 0-9-1（高级消息队列协议）是一种消息传递协议，可使符合标准的客户端应用程序与符合标准的messaging middleware brokers进行通信。

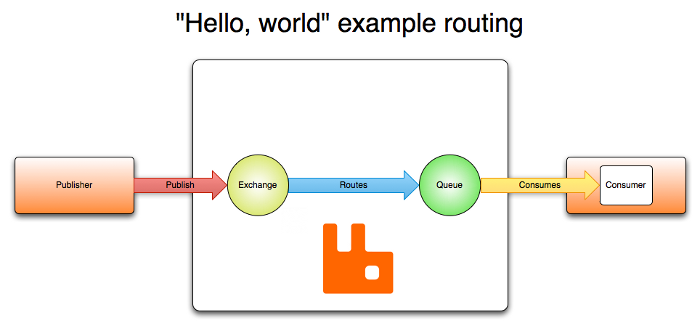
**[Brokers and Their Role](http://www.rabbitmq.com/tutorials/amqp-concepts.html" \l "brokers-role)**

Messaging brokers从*publishers* （发布它们的应用程序，也称为生产者）receive messages并将消息route给消费者（处理它们的应用程序）。

由于它是一种网络协议，publishers, consumers和broker都可以驻留在不同的机器上。

**[AMQP 0-9-1 Model in Brief](http://www.rabbitmq.com/tutorials/amqp-concepts.html" \l "amqp-model)**

AMQP 0-9-1模型具有以下世界观：消息发布到*exchanges*，通常与post offices或mailboxes进行比较。 Exchange然后使用称为*bindings*的规则将消息副本分发到队列。 然后，AMQP brokers将消息传递给订阅了队列的消费者，或者消费者根据需要从队列中fetch/pull消息。



publishing消息时，publishers可以指定各种消息属性 (message meta-data)。 broker可以使用这些元数据中的一部分，但其余部分对broker完全不透明，只能由接收消息的应用程序使用。

网络不可靠，应用程序可能无法处理消息，因此AMQP模型具有消息确认的概念：当消息传递给消费者时，消费者会自动或在应用程序开发人员选择这样做时通知*broker*。 当消息确认正在使用中时，*broker*只会在收到该消息（或消息组）的通知时从消息队列中完全删除消息。

在某些情况下，例如，当消息无法路由时，消息可能会返回给发布者，丢弃，或者如果broker实现扩展，则会放入所谓的"dead letter queue" (死信队列)中。 发布者通过使用某些参数发布消息来选择如何处理这种情况。

Queues, exchanges and bindings统称为AMQP *entities*。

**AMQP is a Programmable Protocol**

AMQP 0-9-1是一种可编程协议，AMQP 0-9-1 entities and routing schemes主要由应用程序本身定义，而不是broker administrator。 因此，规定了protocol operations，declare queues和exchanges，定义它们之间的bindings，subscribe queues等等。

这为应用程序开发人员提供了很大的自由度，但也要求他们意识到潜在的定义冲突。 实际上，定义冲突很少见，常常表明配置错误。

应用程序声明他们需要的AMQP 0-9-1 entities，定义必要的routing schemes，并可以选择删除不再使用的AMQP 0-9-1 entities。

**[Exchanges and Exchange Types](http://www.rabbitmq.com/tutorials/amqp-concepts.html" \l "exchanges)**

*Exchanges* 是发送消息的AMQP实体。 *Exchanges* 会接收一条消息并将其route到零个或多个队列中。 使用的路由算法取决于*exchange type*和称为rules called *bindings*.。 AMQP 0-9-1 brokers提供四种exchange types：

|  |  |
| --- | --- |
| **Name** | **Default pre-declared names** |
| Direct exchange | (Empty string) and amq.direct |
| Fanout exchange | amq.fanout |
| Topic exchange | amq.topic |
| Headers exchange | amq.match (and amq.headers in RabbitMQ) |

除了exchange type之外，exchanges还用许多属性声明，其中最重要的是：

* Name
* Durability (exchanges survive broker restart)
* Auto-delete (exchange is deleted when last queue is unbound from it)
* Arguments (optional, used by plugins and broker-specific features)

Exchanges可以durable or transient。 broker重启后Durable exchanges仍然存在，而transient exchanges不会（当broker重新联机时必须重新申报）。 并非所有场景和用例都需要exchanges才能durable。

**[Default Exchange](http://www.rabbitmq.com/tutorials/amqp-concepts.html" \l "exchange-default)**

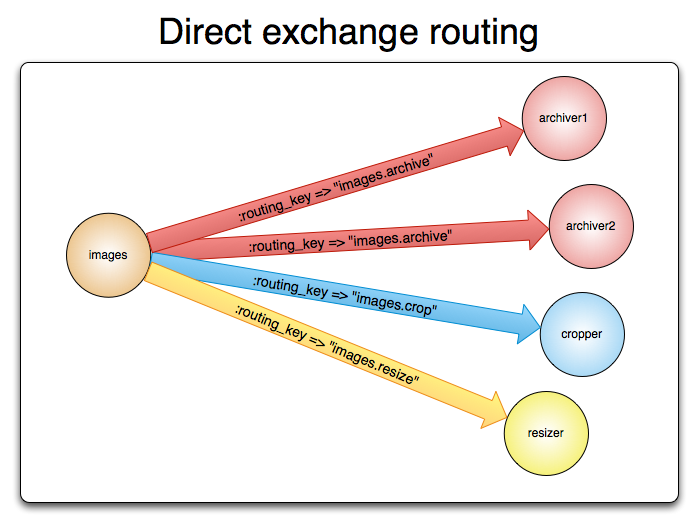
default exchange是broker预先声明的没有名称(empty string)的direct exchange。 它有一个特殊的属性，对于简单的应用程序非常有用：创建的每个队列都使用与队列名称相同的routing key自动绑定到它。

例如，当您声明名称为“search-indexing-online”的队列时，AMQP 0-9-1 broker将使用“search-indexing-online”作为routing key将其绑定到default exchange。 因此，发送到具有路由键“search-indexing-online”的default exchange的消息将被路由到队列“search-indexing-online”。 换句话说，默认的exchange使得它似乎有可能将消息直接传递给队列，尽管这在技术上并非如此。

**[Direct Exchange](http://www.rabbitmq.com/tutorials/amqp-concepts.html" \l "exchange-direct)**

direct exchange基于message routing key将消息delivers给队列。 direct exchange对于消息的单播路由是理想的（尽管它们也可用于多播路由）。 下面是它的工作原理：

* queue 通过routing keybinds到exchange
* 当具有routing keyR的新消息到达direct exchange时，如果K=R,则exchange将其route到队列，Direct exchange通常用于以循环方式在多个workers(同一应用程序的实例)之间分配任务。当这样做时，重要的是要明白，在AMQP 0-9-1中，消息在consumer之间而不在队列之间进行负载均衡。

direct exchange 可以用图形表示如下：

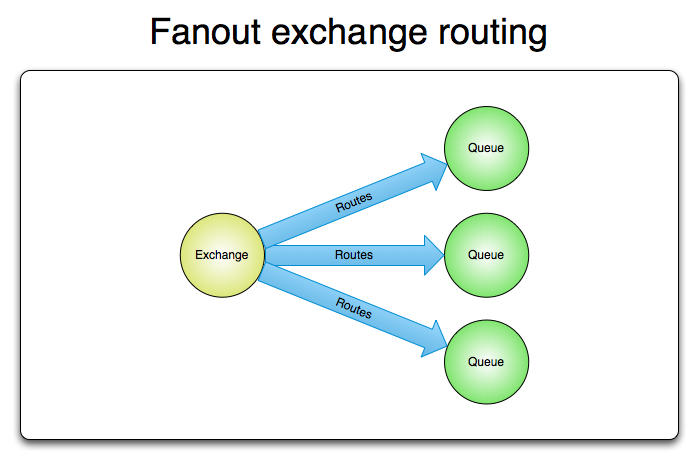
**[Fanout Exchange](http://www.rabbitmq.com/tutorials/amqp-concepts.html" \l "exchange-fanout)**

fanout exchange将消息routes到绑定到它的所有队列，并且routing key被忽略。 如果N个队列绑定到fanout exchange，则当向该exchange发布新消息时，将该消息的副本传递到所有N个队列。 fanout exchange是消息广播路由的理想选择。

由于fanout exchange向每个绑定的队列传递消息的副本，因此它的用例非常相似：

* 大型多人在线(MMO)游戏可将其用于排行榜更新或其他global events
* 体育新闻网站可以使用fanout exchange来近乎实时地向移动客户端发布分数更新
* Distributed systems 可以broadcast 各种状态和配置更新
* 群聊可以使用fanout exchange在参与者之间分发消息(尽管AMQP没有内置的在线概念，因此XMPP可能是更好的选择)

Fanout exchange 可以用图形表示如下：



**[Topic Exchange](http://www.rabbitmq.com/tutorials/amqp-concepts.html" \l "exchange-topic)**

Topic exchanges根据routing key和用于将队列绑定到exchange的模式之间的匹配，将消息路由到一个或多个队列。 Topic exchanges类型通常用于实现各种publish/subscribe模式变体。 Topic exchanges通常用于消息的多播路由。

Topic exchanges具有非常广泛的用例。 每当有问题涉及多个consumers/applications有选择地选择他们想要接收哪种类型的消息时，应考虑使用topic exchanges。

示例用途:

* 分发与特定地理位置相关的数据，例如销售点
* 后台任务处理有多个workers完成，每个worker都能够处理特定的一组任务
* 股票价格更新(以及其他类型财务数据的更新)
* 设计分类或标记的新闻更新(例如,仅针对特定运动或团队)
* 协调cloud中的不同种类的服务
* Distributed architecture/OS-specific software builds or packaging ，每个bulder只能处理一个体系结构或操作系统

**[Headers Exchange](http://www.rabbitmq.com/tutorials/amqp-concepts.html" \l "exchange-headers)**

headers exchange是为多个属性进行路由而设计的，这些属性比routing key更容易表示为message headers。 Headers exchanges忽略routing key属性。 相反，用于路由的属性取自headers属性。 如果头部的值等于绑定时指定的值，则认为消息匹配。

可以使用多个header来将队列绑定到headers exchange以进行匹配。 在这种情况下，broker需要来自应用程序开发人员的另一条信息，即是否应该考虑具有任何匹配的消息或全部消息？ 这就是“x-match”绑定参数的用途。 当“x-match”参数设置为“any”时，只有一个匹配的header值就足够了。 或者，将“x-match”设置为“all”则强制所有值必须匹配。

Headers exchanges可以被视为"direct exchanges on steroids"。 由于它们基于header values进行route，所以它们可以用作direct exchanges，其中routing key不一定是字符串; 例如，它可能是一个integer或hash(dictionary)。

**[Queues](http://www.rabbitmq.com/tutorials/amqp-concepts.html" \l "queues)**

AMQP 0-9-1模型中的队列与其他message- and task-queueing systems中的队列非常相似：它们存储应用程序使用的消息。 Queues与exchanges共享一些属性，但也有一些附加属性：

* Name
* Durable (the queue will survive a broker restart)
* Exclusive (used by only one connection and the queue will be deleted when that connection closes)
* Auto-delete (queue that has had at least one consumer is deleted when last consumer unsubscribes)
* Arguments (optional; used by plugins and broker-specific features such as message TTL, queue length limit, etc)

在可以使用队列之前，必须声明它。 如果它不存在，声明一个队列将导致它被创建。 如果队列已经存在并且其属性与声明中的属性相同，则声明不起作用。 当现有的队列属性与声明中的不同时，将引发代码为406（PRECONDITION\_FAILED）的通道级异常。

**[Queue Names](http://www.rabbitmq.com/tutorials/amqp-concepts.html" \l "queue-names)**

应用程序可能会选择队列名称或要求broker为其生成名称。 队列名称最多可以有255个字节的UTF-8字符。 AMQP 0-9-1 broker可以代表应用程序生成唯一的队列名称。 要使用此功能，请传递一个空字符串作为队列名称参数。 生成的名称将通过队列声明响应返回给客户端。

Queue names以"amq."开头。 由broker保留供内部使用。 尝试声明具有违反此规则的名称的队列将导致出现通道级别的异常，并显示回复代码403（ACCESS\_REFUSED）。

**[Queue Durability](http://www.rabbitmq.com/tutorials/amqp-concepts.html" \l "queue-durability)**

Durable queues被持久化到磁盘，从而在broker重新启动时幸存下来。非durable的队列称为瞬态(transient)。 并非所有场景和用例都要求队列持久。

队列的Durability不会使routed到该队列的消息持久。 如果broker被取下然后重新启动，则durable queue将在broker启动期间re-declared，但是只有持久消息才会被恢复。

**[Bindings](http://www.rabbitmq.com/tutorials/amqp-concepts.html" \l "bindings)**

Bindings是exchanges使用（除其他外）将消息route到queues的规则。 为了指示exchange E将消息route到队列Q，Q必须被绑定到E.绑定可以具有由某些exchange types使用的可选*routing key* attribute。 routing key的目的是选择发布到exchange的某些消息，将其routed到bound queue。 换句话说，routing key就像一个filter。

做一个比喻:

* Queue 就像你在纽约市的目的地
* Exchange 就像肯尼迪机场
* Bindings 是从肯尼迪到目的地的路线。可以有零活许多方法来到达它

具有这种间接层使得路由方案是不可能的或者很难通过直接发布到队列来实现，并且还消除了开发人员必须做的一定量的重复工作应用。

如果AMQP消息无法路由到任何队列（例如，因为没有队列绑定到exchange），它将被丢弃或返回给发布者，具体取决于发布者设置的消息属性。

**[Consumers](http://www.rabbitmq.com/tutorials/amqp-concepts.html" \l "consumers)**

将消息存储在队列中是无用的，除非应用程序可以使用它们。 在AMQP 0-9-1模型中，应用程序执行此操作有两种方法：

* 将消息delivered给他们----推送 ("push API")
* 根据需要拉去messages----拉去 ("pull API")

使用"push API"，应用程序必须指示消费来自特定队列的消息。 当他们这样做时，我们说他们注册了一个消费者，或者简单地说，订阅一个队列。 每个队列可能有多个消费者，或者注册一个独占消费者（消费时排除队列中的所有其他消费者）。

每个消费者（subscription）都有一个称为*consumer tag*的标识符。 它可以用来取消订阅消息。 消费者标签只是字符串。

**[Message Acknowledgements](http://www.rabbitmq.com/tutorials/amqp-concepts.html" \l "message-acknowledge)**

Consumer applications - receive and process messages的应用程序 - 有时可能无法处理单个消息，或者有时会崩溃。 网络问题也有可能导致问题。 这引出了一个问题：AMQP broker何时应该从队列中移除消息？ AMQP 0-9-1规范提出了两种选择：

* broker 向应用程序发送消息后(使用basic.deliver或basic.get-okAMQP方法)
* After the application sends back an acknowledgement (using basic.ack AMQP method).

前者称为automatic acknowledgement model，后者称为explicit acknowledgement model。 通过explicit model，应用程序可以选择何时send an acknowledgement。 在接收到消息之后，或者在处理之前或在完全处理消息（例如，成功获取网页，处理并将其存储到某个持久数据存储区中）之后将其保留到数据存储区之后，它可以是正确的。

如果消费者在未发送确认的情况下死亡，则AMQP broker将重新发送给其他消费者，或者如果当时没有可用消费者，broker将等待至少一个消费者注册相同的队列，然后再尝试重新redelivery。

**[Rejecting Messages](http://www.rabbitmq.com/tutorials/amqp-concepts.html" \l "messages-reject)**

当consumer application receives message时，该消息的处理可能会成功，也可能不会成功。 应用程序可以通过拒绝消息向broker指出消息处理失败（或当时无法完成）。 当拒绝消息时，应用程序可以要求broker丢弃或重新发送消息。 当队列中只有一个消费者时，确保您不会通过一次又一次地拒绝并重新发送来自同一个消费者的消息来创建无限的消息传递循环。

**[Negative Acknowledgements](http://www.rabbitmq.com/tutorials/amqp-concepts.html" \l "messages-nack)**

使用basic.reject AMQP方法拒绝消息。 basic.reject有一个限制：无法像确认一样拒绝多条消息。 但是，如果您使用的是RabbitMQ，那么有一个解决方案。 RabbitMQ提供AMQP 0-9-1扩展，称为*negative acknowledgements* or *nacks*。 有关更多信息，请参 [the help page](http://www.rabbitmq.com/nack.html)。

**[Prefetching Messages](http://www.rabbitmq.com/tutorials/amqp-concepts.html" \l "messages-prefetch)**

对于多个消费者共享队列的情况，能够指定在发送下一个确认之前，每个消费者可以一次发送多少消息是有用的。 这可以用作简单的负载平衡技术，或者如果消息倾向于批量发布，则可以提高吞吐量。 例如，如果生产应用程序每分钟发送一次消息都是因为它正在执行的工作的性质。.

请注意，RabbitMQ仅支持channel-level prefetch-count，而不支持基于connection或基于size的prefetching。

**[Message Attributes and Payload](http://www.rabbitmq.com/tutorials/amqp-concepts.html" \l "messages)**

AMQP模型中的消息具有属性。 一些属性非常常见，以至于AMQP 0-9-1规范定义了它们，应用程序开发人员不必考虑确切的属性名称。 一些例子是

* Content type
* Content encoding
* Routing key
* Delivery mode (persistent or not)
* Message priority
* Message publishing timestamp
* Expiration period
* Publisher application id

某些属性由AMQP brokers使用，但大多数属性可以接收它们的应用程序解释。 有些属性是可选的，称为*headers*。 它们与HTTP中的X-Headers类似。 message published设置message属性。

AMQP消息也有一个*payload* （它们携带的数据），AMQP brokers将其视为一个不透明的byte array。 broker不会检查或修改payload。 消息可能只包含属性而没有payload。 使用JSON，Thrift，Protocol Buffers和MessagePack等序列化格式来序列化结构化数据以便将其发布为消息payload是很常见的。 AMQP同伴通常使用"content-type" 和 "content-encoding"字段来传达这些信息，但这只是惯例而已。

Messages可能会作为持久性发布，这会使AMQP broker将它们保存到磁盘。 如果服务器重新启动，系统会确保接收到的持久性消息不会丢失。 简单地将消息发布到durable exchange或者将其发送到队列的事实是持久的并不会使消息持久化：这完全取决于消息本身的持久模式。 将消息发布为持久性会影响性能（就像使用数据存储一样，持久性在性能上会带来一定的成本）。

**[Message Acknowledgements](http://www.rabbitmq.com/tutorials/amqp-concepts.html" \l "acknowledgements)**

由于网络不可靠并且应用程序失败，通常需要获得某种processing acknowledgement。 有时只需要确认已收到消息的事实。 有时，确认意味着消息经过验证并由消费者处理，例如，验证为具有强制性数据并持久保存到数据存储或编入索引。

这种情况非常普遍，所以AMQP 0-9-1有一个内置的功能，称为*message acknowledgements*（有时称为*acks*），消费者用它来确认消息delivery和/或processing。 如果应用程序崩溃（AMQP broker在连接关闭时发现此事件），如果预期对消息的确认但未被AMQP broker收到，则消息将重新排队（并可能立即传送给其他消费者，如果有的话）存在）。

通过协议内置的acknowledgements功能可帮助开发人员构建更强大的软件。

**[AMQP 0-9-1 Methods](http://www.rabbitmq.com/tutorials/amqp-concepts.html" \l "amqp-methods)**

AMQP 0-9-1被构建为多种方法。 方法是操作（如HTTP方法），与面向对象编程语言中的方法没有任何共同之处。 AMQP方法被分组到类中。 类只是AMQP方法的逻辑分组。  [AMQP 0-9-1 reference](http://www.rabbitmq.com/amqp-0-9-1-reference.html)资料包含所有AMQP方法的全部细节。

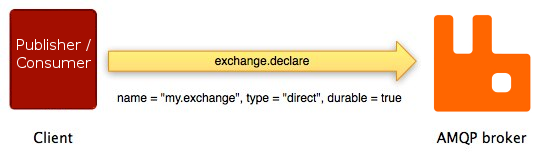
让我们来看看*exchange* class，这是一组与exchanges运营相关的方法。 它包括以下操作：

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

（请注意，RabbitMQ站点参考还包括RabbitMQ-specific的exchange class扩展，我们将不在本指南中讨论）。

上述操作形成逻辑对：exchange.declare和exchange.declare-ok，exchange.delete和exchange.delete-ok。 这些操作是“requests”（由clients发送）和“responses”（由brokers response上述“requests”发送）。

作为一个例子，client要求broker使用exchange.declaremethod申报新的exchange：



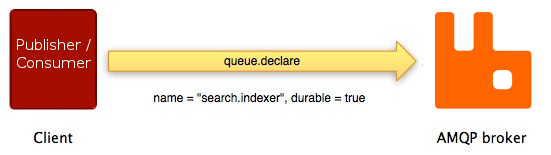
如上图所示，exchange.declare带有几个参数。 它们使client能够指定exchange name，type, durability flag等。

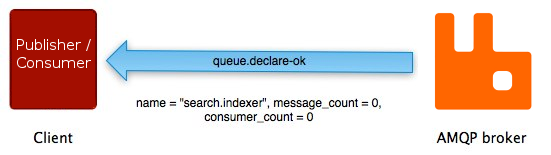
如果操作成功，broker使用exchange.declare-ok方法进行响应：



exchange.declare-ok除channel number外不带任何参数（通道将在本指南稍后介绍）。

对于AMQP队列类中的另一个方法对，事件序列非常相似：queue.declare和queue.declare-ok：





并非所有的AMQP方法都有相应的方法。 有些（basic.publish是使用最广泛的）没有相应的“response”方法，而其他一些（例如basic.get）有多个可能的“response”。

**[Connections](http://www.rabbitmq.com/tutorials/amqp-concepts.html" \l "amqp-connections)**

AMQP连接通常是长期存在的。 AMQP是一种使用TCP进行可靠传输的应用程序级协议。 AMQP连接使用身份验证，并且可以使用TLS（SSL）进行保护。 当应用程序不再需要连接到AMQP broker时，它应该正常关闭AMQP connection，而不是突然关闭底层TCP connection。

**[Channels](http://www.rabbitmq.com/tutorials/amqp-concepts.html" \l "amqp-channels)**

某些应用程序需要多个连接到AMQP broker。 但是，不希望同时打开许多TCP连接，因为这样做会占用系统资源并使配置防火墙变得更加困难。 AMQP 0-9-1连接与可被认为是"lightweight connections that share a single TCP connection"。

对于使用多个threads/processes进行处理的应用程序，通常每个threads/processes都打开一个新channel，而不共享它们之间的channel。

特定通道上的通信与另一个通道上的通信完全分离，因此每个AMQP方法都会携带一个clients用来确定该方法适用于哪个channel number（例如，因此需要调用哪个事件处理程序）。

**[Virtual Hosts](http://www.rabbitmq.com/tutorials/amqp-concepts.html" \l "amqp-vhosts)**

为了使单个broker可以托管多个孤立的“environments”（groups of users，exchanges, queues等），AMQP包含*virtual hosts* (vhosts)的概念。 它们与许多流行的Web服务器使用的虚拟主机相似，并提供AMQP实体所处的完全隔离的环境。 AMQP客户端指定在AMQP连接协商期间他们想要使用哪些虚拟主机。

**[AMQP is Extensible](http://www.rabbitmq.com/tutorials/amqp-concepts.html" \l "amqp-extensibility)**

AMQP 0-9-1有几个扩展点：

* [Custom exchange types](http://www.rabbitmq.com/devtools.html#miscellaneous) 可让开发人员实现routing方案，即exchange提供的即用类型的exchange type，例如geodata-based routing
* exchanges and queues 可以包含broker可以使用的其他属性。例如，RabbitMQ中的[per-queue message TTL](http://www.rabbitmq.com/ttl.html)以这种方案实现。
* Broker-specific 的协议扩展. 例如：请参阅[extensions that RabbitMQ implements](http://www.rabbitmq.com/extensions.html)
* 可引入[New AMQP 0-9-1 method classes](http://www.rabbitmq.com/amqp-0-9-1-quickref.html#class.confirm) .
* 可以使用 [additional plugins](http://www.rabbitmq.com/plugins.html)对broker进行扩展，例如， [RabbitMQ management](http://www.rabbitmq.com/management.html)前端和HTTP API作为插件实现。

这些功能使AMQP 0-9-1模型更加灵活，适用于各种各样的问题。

**[AMQP 0-9-1 Clients Ecosystem](http://www.rabbitmq.com/tutorials/amqp-concepts.html" \l "amqp-clients)**

许多流行的编程语言和平台都有[many AMQP 0-9-1 clients](http://www.rabbitmq.com/devtools.html)。 他们中的一些人密切关注AMQP术语，仅提供AMQP方法的实现。 一些其他人有更多的功能，便利的方法和抽象。 一些客户端是异步（非阻塞），一些是同步（阻塞），一些支持这两种模式。 某些客户端支持特定于供应商的扩展（例如，RabbitMQ特定扩展）。

由于AMQP的主要目标之一是互操作性，因此开发人员理解协议操作并不是限制特定客户端library的术语。 通过这种方式与使用不同library的开发人员进行交流将更加容易。