RabbitMQ

2021年12月1日 8:51

https://www.rabbitmq.com/tutorials/tutorial-one-java.html

==========

1.消息持久化

2.ACK确认机制

3.设置集群镜像模式

4.消息补偿机制

解耦，异步，削(xue)峰

缺点：降低可用性(依赖了MQ，如果MQ宕机)，增加复杂性

==========

==========

https://www.cnblogs.com/zhanxiaomi/p/14072601.html

在3个结点都可能出现消息丢失

1. 生产者，向MQ发送消息时丢失了消息，可能是由于网络原因
2. MQ，接收到了消息，在没有持久化到磁盘，也没有被消费掉的时候，MQ宕机。
3. 消费者，从MQ拉取到了消息，但没有真正消费完。由于已被拉取，所以MQ认为已消费。

生产者没有成功把消息发送到MQ
2个解决方案，事务机制和confirm机制，后者常用。

事务机制

RabbitMQ提供了事务功能，生产者发送数据前开启事务channel.txSelect。然后发送消息，如果消息没有成功被RabbitMQ接收到，那么生产者会收到异常报错，此时就可以回滚事务channel.txRollback；如果收到了消息，那么可以提交事务channel.txCommit。

confirm机制

RabbitMQ可以开启 confirm 模式，在生产者那里设置开启 confirm 模式后，生产者每次写入的消息都会被分配一个唯一的id，如果消息成功写入RabbitMQ，RabbitMQ会给生产者一个 ack 消息。如果RabbitMQ 没能处理这个消息，会回调你的一个 nack 接口。而且你可以结合这个机制 在自己的内存中维护 每个消息id的状态，如果超过一段时间还没有收到回应，那么可以重发。

事务机制是同步的，消耗性能，降低吞吐量；
confirm 是异步的，生产者发送消息后，不需要等待RabbitMQ的回调，就可以发送下一条消息，当RabbitMQ成功接收到消息后，会自动异步回调生产者的接口返回是否成功。
。。这个接口怎么给RabbitMQ

RabbitMQ 收到消息后丢失
解决方案，开启RabbitMQ的 持久化。 当生产者把消息成功写入RabbitMQ后，RabbitMQ就把消息持久化到磁盘，结合上面的confirm机制，只有当消息成功持久化到磁盘后，才会回调生产者的接口返回ack 消息。
持久化配置

创建queue的时候将其设置为持久化(durable设置为true)，这样就可以保证RabbitMQ持久化queue的 元数据，但是它不会持久化queue里的消息

发送消息的时候将消息的 deliveryMode 设置为2，这就是将消息设置为 持久化的，RabbitMQ会将消息持久化到磁盘上。

这2个持久化要同时设置。
(。。mirrored-queue即镜像队列)

消费者丢失消息
如果RabbitMQ 成功把消息发送给 消费者，那么RabbitMQ的ack 机制就会自动返回成功，表明发送消息成功，以后不会再发送这个消息；如果此时，消费者还没处理完，宕机，那么这个消息就丢失了。
解决方案，关闭RabbitMQ的 自动ack，而是消费者在处理完后，调用ack。

防止重复消费。
由于消费者在处理完后，发送ack 给RabbitMQ。如果这个ack 没有发送到 MQ，那么MQ会 认为这条消息没有被消费过，会再次分发给其他消费者。
解决方案：保证消息的唯一性，就算多次传输，不然让消息的多次消费带来影响；保证幂等性；

在生产消息时，MQ内部为每条生产者发送的消息生成一个inner-msg-id，作为去重和幂等的依据(消息投递失败并重传)，避免重复消费进入队列。

在消费消息时，要求消息体中必须有一个id 作为去重和幂等的依据，避免同一条消息被重复消费。

如果消息是数据库的insert，给这个消息做一个主键，这样insert后，后续的insert会主键冲突。

如果是redis 的set，不需要解决，因为set操作本身就是幂等的。

准备一个第三方 来做消费记录，以redis为例，给消息分配一个全局id，只要消费

过，就将<id,message> 以k-v形式 写入到redis；消费者消费前，先去redis中查询消费记录。

==========

==========

==========

http://rabbitmq.mr-ping.com/description.html

消息系统允许 软件，应用 相互连接和扩展，这些应用可以相互连接起来组成 一个更大的应用，或者将 用户设备和数据进行连接，消息系统通过将 消息的发送和接受分离 来实现应用程序的 异步和 解耦。

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

可靠性
RabbitMQ提供多种技术 可以让你在 性能和可靠性之间 进行权衡。这些技术包括 持久性机制，投递确认，发布者证实 和 高可用机制。

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

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

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

型。

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

多协议
RabbitMQ支持多种消息协议的消息传递

广泛的客户端
覆盖(所有的)编程语言

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

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

插件系统
RabbitMQ 附带了各种插件来对自己进行扩展，你甚至可以写自己的插件

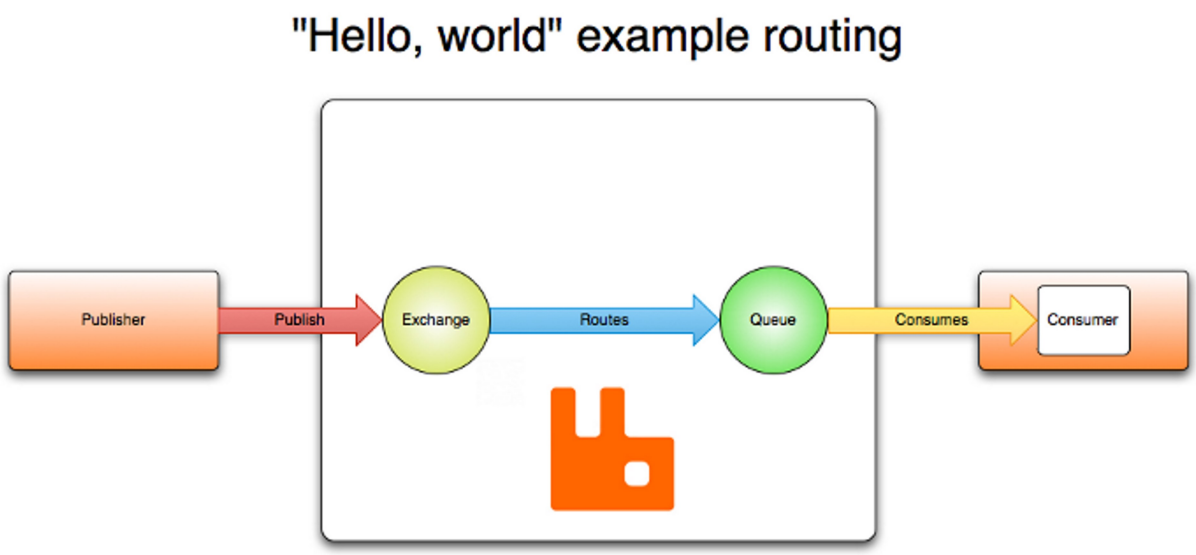
安装

AMQP 0.9.1 模型解析
AMQP(高级消息队列协议) 是一个网络协议。它支持符合要求的客户端应用(application)和 消息中间件代理(messaging middleware broker) 之间进行通信。

消息代理和他们所扮演的角色
消息代理(message brokers) 从发布者(publishers) 也称为生产者(producers) 那里接收消息，并根据既定的路由规则把接收到的消息发送到处理消息的消费者(consumers)

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

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



发布者(publisher)发布消息时 可以给消息指定 各种消息属性(message meta-data)。有些属性可能会被消息代理(broker)使用，然而其他的属性则是完全不透明的，它们只能被接受消息的应用使用。

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

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

队列，交换机和绑定 统称为 AMQP实例(AMQP entities)

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

应用程序 声明AMQP实体，定义需要的路由方案，或者删除不需要的AMQP实体。

交换机和交换机类型
交换机是用来发送消息的AMQP实体。交换机拿到一个消息后，将它路由给一个 或 0个 队列。它使用哪种路由算法 是由交换机类型 和 被称为绑定(bindings) 的规则所决定的。
AMQP 0-9-1 的代理提供了4种交换机。

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

Headers exchange(头交换机)	amq.match(and amq.headers in RabbitMQ)

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

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

交换机可以有2个状态：持久(durability)，暂存(transient)。持久化的交换机会在 消息代理 重启后依然存在，而暂存的交换机则不会(它们需要在 代理再次上线后 重新被声明)。然后并不是所有的应用场景都需要持久化的交换机。

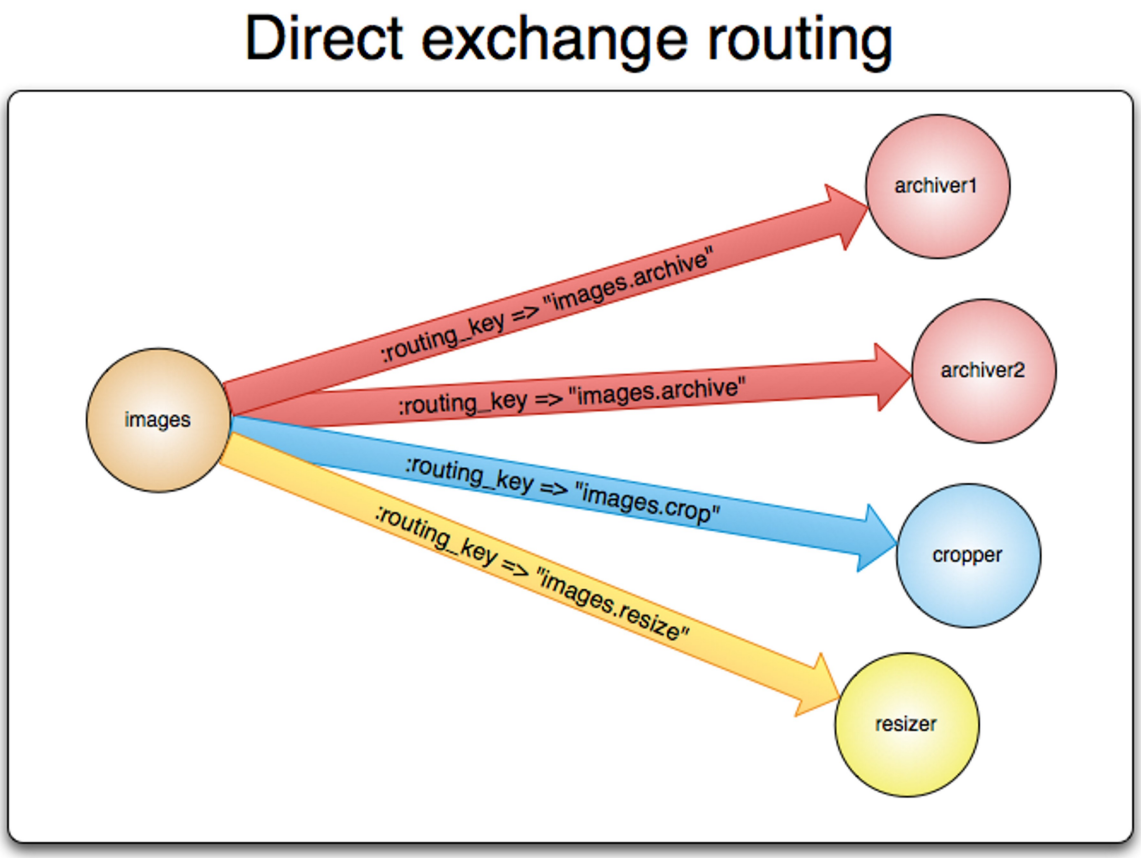
默认交换机
默认交换机(default exchange) 实际上是一个 由消息代理 预先声明好的 没有名字(名字为空字符串) 的直接交换机(direct exchange)。 它有一个特殊的属性 使得它对于简单应用 特别有用处：那就是每个新建队列(queue) 都会自动绑定到 默认交换机上，绑定的路由键(routing key) 名称与队列名称相同。

例如：当你声明了一个名为"search-indexing-online"的队列，AMQP代理会自动将其绑定到默认交换机上，绑定的路由键名称也是"search-indexing-online"。因为，当携带者名为"search-indexing-online"的路由键的 消息被发送到 默认交换机时，此消息会被默认交换机路由到 名为"search-indexing-online"的队列中。换句话说，默认交换机看起来貌似能够直接将消息投递给队列，尽管技术上并没有做相关的操作。

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

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

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



扇型交换机

fanout exchange 将消息路由给 绑定到它身上的所有队列，而不理会 绑定的路由键。

如果有N个队列绑定到某个扇型交换机上，当有消息发送给此扇型交换机时，交换机会将消息的拷贝分别发送给这所有的N个队列。扇型交换机用来处理 消息的 广播路由(broadcast routing)。

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

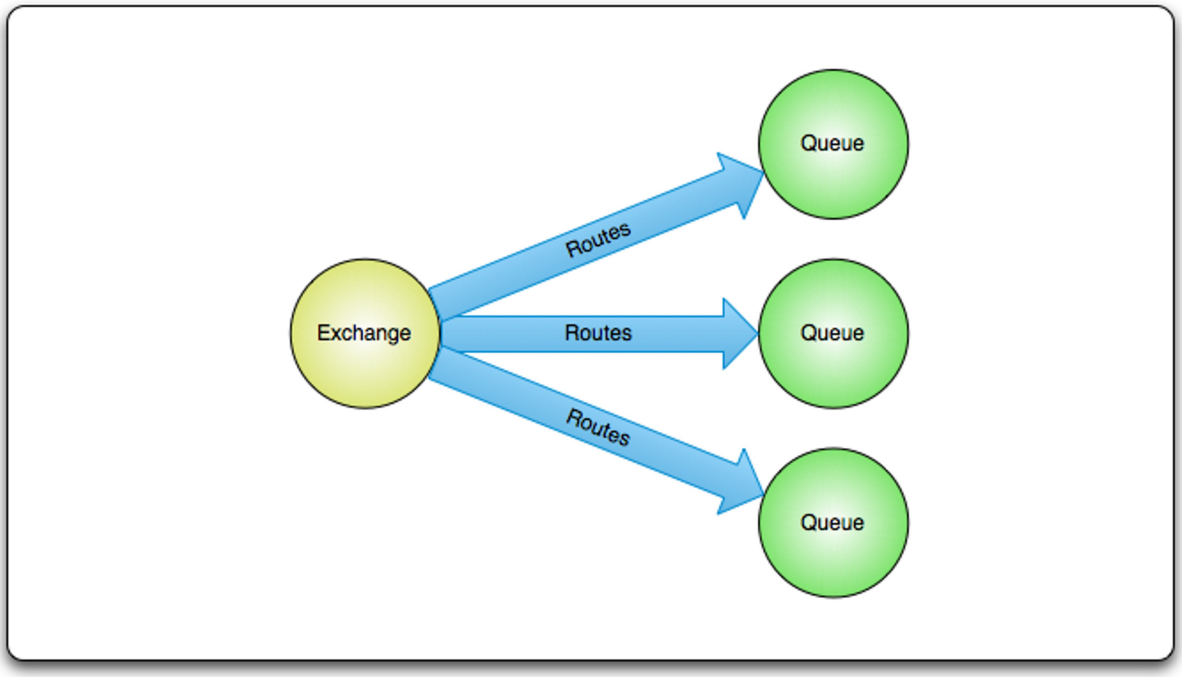
大规模用户在线(MMO)游戏可以用它来处理排行榜更新等全局事件

体育新闻网站可以用它来近乎实时地将比分更新 分发给移动客户端

分发系统 使用它来广播各种状态和配置更新

在群聊的时候，它被用来分发消息给参与群聊的用户 (AMQP 没有内置 presence的概念，因此XMPP 可能会是更好的选择)

Fanout exchange routing



主题交换机

topic exchange 通过对消息的路由键 和 队列到交换机的绑定模式 之间的匹配，将消息路由给 一个或 多个队列。

主题交换机经常用来实现 各种 分发/订阅 模式及其变种。

主题交换机通常用来实现消息的多播路由(multicast routing)

主题交换机有着 非常广泛的用户案例。无论何时，当一个问题涉及到 那些想要 有针对性地选择 需要接受消息的 多消费者/多应用 (multiple consumers/applications) 的时候，主题交换机都可以被列入考虑范围。

使用案例：

分发有关于特定地理位置的数据，例如销售点

由多个工作者完成的 后台任务，每个工作者负责处理某些特定的任务

股票价格更新

涉及到分类或标签的新闻更新

云端的不同种类服务的协调

分布式架构/基于系统的软件封装，其中每个构建者只能处理一个特定的结构或系统。

头交换机

有时，消息的路由操作会 涉及到多个属性，此时 使用消息头就比 用路由键更容易表达，头交换机就是 为此而生的。

头交换机使用多个消息属性来代替路由键 建立路由规则。通过判断消息头的值 能否与 指定的 绑定匹配 来 确定 路由规则。

我们可以绑定一个队列到 头交换机上，并给他们之间的绑定 使用多个用于匹配的头。 这种案例中，消息代理 得从 应用开发者 那里获得 更多一段信息，换句话说，它需要考虑某条消息 是需要 部分匹配还是 全部匹配，上面说的 更多一段信息，就是 'x-match' 参数，当设置为 any时，消息头的任意一个值 被匹配 就满足条件， 当设置为 all时，就需要消息头

的所有值 都匹配成功。
头交换机可以视为 直连交换机的 另一种表现形式。头交换机能够像直连交换机一样工作，不同之处在于 头交换机 的路由规则是建立在 头属性值上的，而不是路由键。路由键必须是一个 字符串，而头属性值没有这个约束，它甚至可以是 整数 或hash值 等。
。。不过这里说的是 匹配， 感觉像是 pattern。。而不是直接 相等。。

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

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

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

队列名称
队列的名字可以由 应用来取，也可以让 消息代理 直接生成一个。
队列的名字可以是 最多255字节的一个 utf-8 字符串。如果希望 AMQP 消息代理生成队列名，需要给队列的name参数 赋值一个空字符串：在同一个 channel 的后续方法中，我们可以使用 空字符串 来表示 之前生成的 队列的名称。之所以之后的方法可以获得正确的队列名是因为channel可以默默记住消息代理 最后一次生成的队列名称。
以"amq." 开始的队列名称 被预留为 消息代理内部使用。如果视图在队列声明中打破这一规则的话，一个channel级别的 403(ACCESS_REFUSED) 会抛出

队列持久化
持久化队列(durable queues) 会被存储在磁盘上，当消息代理(broker)重启时，它依旧存在。没有被持久化的队列被称为 暂存队列(transient queues)。
持久化的队列 并不会 使得 路由到它的消息也具有持久性。 如果消息代理挂了，重新启动，那么重启过程中 持久化队列会被重新声明，无论怎样，只有经过持久化的消息 才能被重新恢复。

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

- 队列 是我们想要去的 位于纽约的目的地
- 交换机 是JFK机场
- 绑定 就是JFK机场 到目的地的路线。能够到达目的地的路线可以是一条或多条。

拥有了交换机 这个中间层，很多由 发布者直接到 队列 难以实现的路由方案 能够得以实现，并且避免了 应用开发者的许多重复劳动。
如果AMQP的消息 无法路由到队列(例如，发送到的交换机没有绑定队列)，消息会被就地销毁或者返还给发布者。如何处理取决于发布者设置的消息属性。

消费者
AMQP 0-9-1模型中，有2种途径可以让 消息被消费：

将消息投递给应用(push API)
应用根据需求主动获取消息(pull API)

使用push API，应用需要明确表示出 它在 某个特定队列中 所感兴趣的，想要消费的消息。如是，我们可以说应用注册了一个消费者，或者说 订阅了一个队列。 一个队列可以注册多个消费者，也可以注册一个 独享的消费者(当独享消费者存在时，其他消费者被排除在外)。
每个消费者(订阅者)都有一个叫做 消费者标签的 标识符。它可以被用来退订消息。消费者标签实际上是一个字符串

消息确认
消费者应用 - 用来接收和处理消息的应用 - 在处理消息的时候偶尔会失败 或者有时 会直接崩溃掉。而且网络原因也有可能引起各种问题。这就给我们出了一个难题，AMQP代理 在什么时候 删除消息才是正确的？ AMQP 0-9-1 规范给我们2种建议：

当消息代理 将消息发送给 应用后立刻删除(使用AMQP方法：basic.deliver 或 basic.get-ok)
待应用 发送一个确认回执 后再删除消息(使用AMQP方法：basic.ack)

前者被称为 自动确认模式，后者被称为 显式确认模式。在显式模式下，由消费者应用来选择什么时候发送确认回执。应用可以在收到消息后立刻发送，或者将未处理的消息存储后发送，或者等到消息被处理完毕后再发送 确认回执
如果一个消费者在 尚未发送确认回执的 情况下 挂掉了，那AMQP代理会将 消息重新投递给另一个消费者。如果当时没有可用的消费者，消息代理会一直等待，知道下一个注册到此队列的消费者，然后尝试投递。

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

Negative Acknowledgements
在AMQP中， basic.reject 方法用来执行 拒绝消息的操作。但 basic.reject 有一个限制：你不能使用它 决绝(..拒绝?)多个带有 确认回执 的消息。但是如果你使用的RabbitMQ，那么你可以使用被称作 negative acknowledgements (也叫 nacks) 的AMQP 0-9-1扩展来解决这个问题。

预期消息
在多个消费者共享一个队列的案例中，明确指定在收到下一个确认回执前 每个消费者 一次可以接受多少条消息 是非常有用的。这可以在试图批量发布消息的时候 起到简单的负载均衡 和 提高消息吞吐量的作用。
注意，RabbitMQ 只支持channel级别的 预期技术，而不是 连接级的 或 基于大小的 预期。

消息属性和有效载荷(消息主体)
AMQP模型中的消息 对象是带有 属性的。有些属性及其常见，以至于 AMQP 0-9-1 明确定义了它们，并且 应用开发者 无需费心思 思考这些属性名字所代表的 具体含义。例如：

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

有些属性被AMQP代理 所使用，但是大多数是 开放给 接受它们的 应用解释器用的。有些属性是可选的 也被称作 消息头。它们和 http协议的 X-Headers 很相似。消息属性需要在消息被发布的时候定义。
AMQP的消息除 属性外，也含有一个有效载荷 - Payload(消息实际携带的数据)，它被AMQP代理当做 不透明的 字节数组 来对待。消息代理不会检查或修改有效载荷。消息可以只包含属性 而不携带 有效载荷。它通常使用类似JSON这种序列化的格式数据，为了节省，协议缓冲器 和 MessagePack 将结构化数据序列化，以便以消息的有效载荷的形式发布。AMQP及其同行们 通常使用 content-type, content-encoding 这2个字段 来辨识 载荷。
消息能够以持久化的方式发布，AMQP代理会将此消息存储在磁盘上。如果服务器重启，系统会确认(..确保?)收到的 持久化消息 未丢失。 简单地将 消息发给 一个 持久化的交换机 或路由给一个持久化的 队列，并不会使得此消息具有 持久化性质：它完全取决于 消息本身的持久模式。将消息以持久化方式发布时，会对性能造成一定的影响。

消息确认
由于网络的不确定性和应用失败的可能性，处理确认回执 就变得非常重要。有时 我们确认消费者 收到消息就可以了，有时 确认回执 意味着 消息已经被验证 并且处理完毕。
这种情况很常见，所以AMQP 091内置了一个 功能 叫做 消息确认(message acknowledgements)，消费者用它来确认 消息已经被 接收或者 处理。
如果一个应用崩溃掉(此时连接会断掉，所以AMQP代理会得知)，而且 消息的确认回执功能已经被开启，但是消息代理尚未获得确认回执，那么消息会被重新放入队列(并且在 还有其它消费者存在 于 此队列的情况下，立即投递给另外一个消费者)

AMQP 0-9-1 方法
AMQP 0-9-1 由许多方法(method)构成。AMQP的方法被分组在 类(class)中。 这里 方法，类 只是 AMQL的分组，和面向对象没有关系。

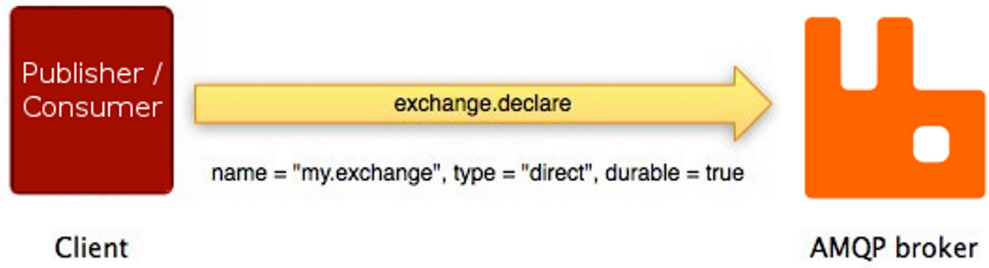
让我们来看看exchange类，有一组方法被关联到 交换机的操作上，这些方法如下所示：

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

RabbitMQ网站参考中， 包含了 特用于 RabbitMQ 的交换机类的扩展，这里并不会讨论。

上面的操作 分为 请求(由客户端发送) 和 响应(由代理发送，用来回应之前的请求)

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

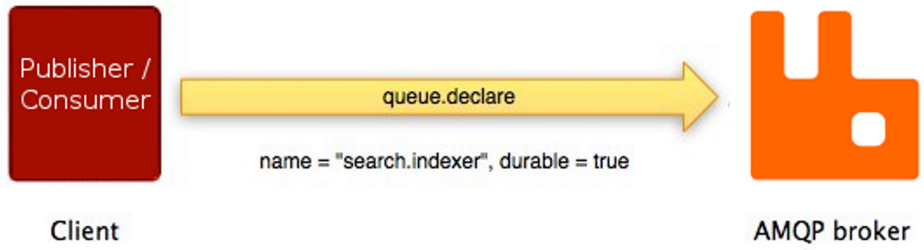


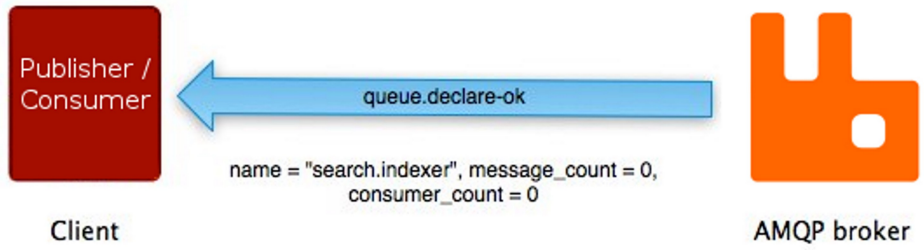
如上图所示，exchange.declare 方法携带好几个参数，这些参数可以允许客户端 指定交换机名称，类型，是否持久化 等。

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

exchange. delclare-ok 方法除了 通道号 之外 没有携带任何其他参数

AMQP queue类 的配对方法 queue.declare 方法 和 queue.declare-ok





不是所有的 AMQP方法 都是成对的。 许多 (basic.publish) 都没有对应的 响应方法，有一些(basic.get) 有着一种以上的 响应方法。

连接

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

通道

有些应用要与AMQP代理 建立多个连接。无论怎样，同时开启多个 TCP连接都是不合适的，因为这样会消耗过多系统资源并且使得 防火墙配置更加困难。AMQP091提供了 channel 来处理 多连接，可以把channel 理解成 共享一个TCP连接的 多个轻量化连接。
在涉及多线程/进程的应用中，为每个线程/进程 开启一个 channel 是很常见的，并且这些 channel 不能被 线程/进程 共享。
一个特定channel 上的通讯 和 其他channel 上的通讯是 完全隔离的，所以 每个AMQP 方法都需要携带一个 channel号，这样客户端就可以 指定此方法 是为哪个channel 准备的。

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

AMQP是可扩展的
有多个扩展点
- 定制化交换机类型
- 交换机和队列的声明中可以包含一些消息代理能够使用的额外属性
- 特定消息代理的协议扩展
- 新的AMQP 091 方法类可被引入
- 消息代理可以被其他的插件扩展

AMQP 0-9-1 快速参考指南

Basic
basic.ack(delivery-tab delivery-tag, bit multiple)
对一条或多条消息进行确认。

basic.cancel(consumer-tag consumer-tag, no-wait no-wait) -> cancel-ok
cancel队列消费者
用来清除消费者。不会影响已经成功投递的消息，但是 会使得 服务器不再将新的消息投送给此消费者。客户端在 发送cancel方法 和 收到cancel-ok回复的 过程中 收到任意数量的消息。

basic.consume(short reserved-1, queue-name queue, consumer-tag consumer-tag, no-local no-local, no-ack no-ack, bit exclusive, no-wait no-wait, table arguments) ➔ consume-ok
告知服务器开启一个 消费者，此消费者实质 是一个 针对 特定队列消息的 持久化请求。消费者在 声明过的channel 中会一直存在，直到客户端清除他们为止。

basic.deliver(consumer-tag consumer-tag, delivery-tag delivery-tag, redelivered redelivered, exchange-name exchange, shortstr routing-key)
将消费者消费通知给客户端
将一条消息通过消费者 投递给客户端。在 异步消息投递模式中，客户端通过 Consume 方法

启动消费者，然后服务器使用 Deliver 方法 将消息送达。

basic.get(short reserved-1, queue-name queue, no-ack no-ack) ➔ get-ok | get-empty
直接访问队列
提供了通过同步通讯的方式 直接访问队列内消息的途径。它针对的是一些 有特殊需求的应用，例如对应用来说 同步的功能性 意义远大于 应用性能。

basic.nack(delivery-tag delivery-tag, bit multiple, bit requeue)
RabbitMQ特有的AMQP扩展
拒绝一条或多条 input 消息
此方法允许客户端拒绝一条或多条 输入消息。它可以用来打断 或清除 大体积 消息的输入，或者将无法处理的消息返回给 消息的原始队列。这个方法也可以在 确认模式(confirm mode) 下被 服务器用来 通知channel 上的消息发布者 有未被处理的消息存在。

basic.publish(short reserved-1, exchange-name exchange, shortstr routing-key, bit mandatory, bit immediate)
发布一条消息 到指定的exchange，消息会通过 配置好的 exchange 根据既定的规则 路由到队列，之后，如果存在事务处理(transaction)，并且事务已经被提交，就会分发给活跃的消费者。

basic.qos(long prefetch-size, short prefetch-count, bit global) ➔ qos-ok
此方法 指定服务的服务质量。QoS 可以分配给当前的channel 或连接内的 所有 channel。qos方法的特定属性 和语义依赖于内容类的 语义。虽然qos方法原则上可以用于服务端及客户端，但是这里的方法 只适用于 服务器端。

basic.recover(bit requeue)
重新投递未被确认的消息
此方法会要求服务器重新投递特定channel内所有未确认的消息。此方法用于替代异步恢复 (async recover)

basic.recover-async(bit requeue)
已弃用，请使用 Recover/Revocer-Ok

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

basic.return(reply-code reply-code, reply-text reply-text, exchange-name exchange, shortstr routing-key)
返回单条处理失败的消息
此方法将发布时带有 immediate 标签的无法投递的，或发布时 带有 mandatory 标签的无法

正确路由的 单条消息返回。应答代码 或文字中会注明失败原因。

Channel

channel.close(reply-code reply-code, reply-text reply-text, class-id class-id, method-id method-id) ➔ close-ok
请求关闭channel
此方法表明发送者希望关闭channel，这通常是 由于内部条件(如强制关闭) 或者由于 处理特定方法引起的错误(即Exception) 触发。当关闭行为是 exception触发时，发送者需要提供引起exception 的方法的 class id 和 method id

channel.flow(bit active) ➔ flow-ok
启用/禁用 对端流
此方法要求 对端 暂停或 重启 消费者 发送的 内容数据流。这是一个 简单的 流控制机制，用来避免信道的队列溢出 或 发现信道接受的消息 是否超出了其处理能力。需要注意的是，此方法的目的不在于控制窗口，它不会影响basic.get-ok 方法返回的内容。

channel.open(shortstr reserved-1) ➔ open-ok
此方法会打开一个信道用于与服务器通讯。

Confirm
此类为RabbitMQ特有的AMQP扩展

confirm.select(bit nowait) ➔ select-ok
此方法用来设置信道以便使用发布者确认回执（acknowledgements）。客户端仅可将此方法用于非事务性信道。

Exchange

exchange.bind(short reserved-1, exchange-name destination, exchange-name source, shortstr routing-key, no-wait no-wait, table arguments) ➔ bind-ok
RabbitMQ特有
此方法将一个交换机绑定到另一个交换机上。

exchange.declare(short reserved-1, exchange-name exchange, shortstr type, bit passive, bit durable, bit auto-delete*, bit internal*, no-wait no-wait, table arguments) ➔ declare-ok
RabbitMQ特有
验证交换机是否存在，不存在就新建，存在就验证 类型是否正确。
RabbitMQ针对AMQP规范实现了一个扩展，此扩展允许将无法正确路由的消息投递到一个替代交换机中（AE）。替代交换机的特性可以帮助判断客户端何时发布了无法路由的消息，并且

能够提供 "or else" 路由语义去对某些消息做特殊处理，其他的消息则由通用方法进行处理。

exchange.delete(short reserved-1, exchange-name exchange, bit if-unused, no-wait no-wait) ➔ delete-ok
此方法用于删除交换机。当一个交换机被删除后，与其绑定的所有队列都会被清除。

exchange.unbind(short reserved-1, exchange-name destination, exchange-name source, shortstr routing-key, no-wait no-wait, table arguments) ➔ unbind-ok
RabbitMQ特有
解除两个交换机之间的绑定关系

Queue
queue.bind(short reserved-1, queue-name queue, exchange-name exchange, shortstr routing-key, no-wait no-wait, table arguments) ➔ bind-ok
此方法用于绑定队列到交换机。队列绑定到交换机之前不会接收到任何消息。在经典消息模型中，存储转发队列绑定到直连交换机，订阅队列绑定到主题交换机。

queue.declare(short reserved-1, queue-name queue, bit passive, bit durable, bit exclusive, bit auto-delete, no-wait no-wait, table arguments) ➔ declare-ok
此方法用于创建或检查队列。当新建一个队列时，客户端可以指定一系列属性用于控制队列的持久性及其内容，还有队列的分享等级。
RabbitMQ为AMQP规范实现了一些扩展，允许队列创建者控制队列各个方面的行为。

每个队列的消息生命周期

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

队列的过期时间

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

queue.delete(short reserved-1, queue-name queue, bit if-unused, bit if-empty, no-wait no-wait) ➔ delete-ok
此方法用于删除一个队列。如果服务器设置了死信队列（dead-letter queue），当队列被某个删除时，任何依存于此队列的消息都会被发送到死信队列中，队列上的所有消费者都会被清除掉。

queue.purge(short reserved-1, queue-name queue, no-wait no-wait) ➔ purge-ok
此方法会将队列中的所有不处于等待 确认回执（acknowledgment）状态的消息全部移除。

queue.unbind(short reserved-1, queue-name queue, exchange-name exchange, shortstr routing-key, table arguments) ➔ unbind-ok
此方法用于解除队列与交换机的绑定关系。

Tx

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

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

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

http://rabbitmq.mr-ping.com/ClientDocumentation/java-api-guide.html

RabbitMQ Java 客户端使用com.rabbitmq.client作为它的顶级包。关键的类和接口有：

Channel: 代表 AMQP 0-9-1通道，并提供了大多数操作（协议方法）。
Connection: 代表 AMQP 0-9-1 连接
ConnectionFactory: 构建Connection实例
Consumer: 代表消息的消费者
DefaultConsumer: 消费者通用的基类
BasicProperties: 消息的属性（元信息）
BasicProperties.Builder: BasicProperties的构建器

通过Channel（通道）的接口可以对协议进行操作。Connection（连接）用于开启通道，注册连接的生命周期内的处理事件，并且关闭不再需要的连接。ConnectionFactory用于实例化Connection对象，并且可以通过ConnectionFactory来进行诸如vhost、username等属性的设置。

```
ConnectionFactory factory = new ConnectionFactory();
// "guest"/"guest" by default, limited to localhost connections
factory.setUsername(userName);
factory.setPassword(password);
factory.setVirtualHost(virtualHost);
factory.setHost(hostName);
factory.setPort(portNumber);
```

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

Property	Default Value
Username	"guest"
Password	"guest"
Virtual host	"/"
Hostname	"localhost"
port	5672 for regular connections, 5671 for connections that use TLS

```
ConnectionFactory factory = new ConnectionFactory();
factory.setUri("amqp://userName:password@hostName:portNumber/virtualHost");
Connection conn = factory.newConnection();
```

默认情况下， guest用户只能用本地进行连接。

Connection接口就可以用来开启通道(Channel)了：

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

通道可用于消息的发送和接收

对通道和连接进行关闭

```
channel.close();
conn.close();
```

虽然将通道关闭掉是最佳实践，但并不是必须的操作。因为无论何种情况，通道都会在底层的连接关闭时自动关闭掉。

客户端connections是长连接。底层协议的设计和优化都考虑到了长连接的需求。这意味着对诸如消息发送之类的每个操作都建立一个连接的形式是极其不推荐的，那样做会产生大量的网络往返和开销。

Channels 虽然也是长期存活的，但是由于有大量的可恢复的协议错误会导致通道关闭，通道的存活期会比连接断一些。虽然每个操作都打开和关闭一个通道不是必须的操作，但是也不是不可行。有的选的情况下，还是优先考虑通道的复用为好。

RabbitMQ 节点可以持有有限的的客户端信息：

客户端的TCP节点（来源IP地址和端口）
使用的凭证

包括RabbitMQ Java客户端在内的AMQP 0-9-1客户端链接可以提供一个自定义标识符，以便在服务器日志 和管理界面中方便地对客户端进行区分。设置好后，日志内容的管理界面中便会对标识符有所体现。标识符即为客户端提供的连接名称。名称可以用于标识应用或应用中特定的组件。虽然名称是可选的，但是强烈建议提供一个，这将大大简化某些操作任务。

RabbitMQ Java 客户端的 ConnectionFactory#newConnection 方法 覆写了）接收客户端提供的连接名称。这是一个修改过的连接样例，用于提供连接名称：

```
ConnectionFactory factory = new ConnectionFactory();
factory.setUri("amqp://userName:password@hostName:portNumber/virtualHost");
// provides a custom connection name
Connection conn = factory.newConnection("app:audit component:event-consumer");
```

交换机和队列的使用

客户端使用用利用交换机和队列这些高级的协议构建块工作。在使用前必须对他们进行声明。简单来讲，对任何一种对象类型进行声明的目的是为了确保它们已经存在，并在需要的时候对其进行创建。

接着上边的例子，以下代码声明了一个交换机以及一个服务端命名的队列，然后将它们绑定到一起。

```
channel.exchangeDeclare(exchangeName, "direct", true);
String queueName = channel.queueDeclare().getQueue();
channel.queueBind(queueName, exchangeName, routingKey);
```

这将会主动声明以下对象，这两个对象都可以使用附加参数进行自定义。但在这里，没有给他们俩定义特殊的参数。

持久化、非自动删除的“直连”形交换机
具有系统生成的名称的，非持久化、独占、自动删除的队列

上边调用的函数使用给定的路由键将队列和交换机绑定起来。

注意，当只有一个客户端打算工作于此队列时，这是一个典型的队列声明方式。队列不需要既定的名称，没有其他客户端使用此队列（独占），队列会被自动清理掉（自动删除）。如果有多个客户端消费打算消费一个既定名称的队列，一下代码更为合适：

```
channel.exchangeDeclare(exchangeName, "direct", true);
channel.queueDeclare(queueName, true, false, false, null);
channel.queueBind(queueName, exchangeName, routingKey);
```

这将会主动进行以下声明：

持久化、非自动删除的“直连”交换机
拥有既定名称的，持久化、非独占、非自动删除的队列

被动声明

队列 和 交换机 可以被动地进行声明。被动声明 会简单地检查提供的名称所对应的实体 是否存在。如果不存在就不会做任何操作。 对于成功检测到的 队列来说， 被动声明会返回和 非被动声明同样的信息，即队列中 处于就绪状态的 消费者 和 消息数量。

如果对于的实体不存在，操作会抛出一个通道级别的异常。然后通道就不能继续使用了，需要打开一个 新的通道。通常在 进行被动声明的 时候 使用临时的一次性通道。

Channel#queueDeclarePassive 和 Channel#exchangeDeclarePassive 方法被用来进行被动声明。下边演示Channel#queueDeclarePassive 的使用：

```
Queue.DeclareOk response = channel.queueDeclarePassive("queue-name");
// returns the number of messages in Ready state in the queue
response.getMessageCount();
// returns the number of consumers the queue has
response.getConsumerCount();
```

Channel#exchangeDeclarePassive 方法的返回值没包含什么有用的信息。只要方法正确返回，并且没有通道异常发生，就意味着交换机已经存在了。

可选响应的操作
一些常见的操作还带有"非等待"版本，这种版本的操作不会等待服务器的响应。例如，下面方法会声明一个队列 并且通知服务器不要发送任何响应

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

"非等待"版本的操作 更具效率，但是安全保障较低，例如，它们更依赖心跳机制去检测失败的操作。如果不确定，就从标准版本的操作用起。"非等待"版本只是在 高级拓扑结构(队列，绑定)的情况下需要。

删除实体和清除消息
可以显式将队列和交换机删除

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

也可以做到当队列为空时对其进行删除：

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

或者当它不再被使用的时候(没有任何消费者对其进行消费)：

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

队列可以被清除(删除里面的所有消息)：

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

发布消息
使用 Channel.basicPublish 将消息发布到交换机中：

```
byte[] messageBodyBytes = "Hello, world!".getBytes();
channel.basicPublish(exchangeName, routingKey, null, messageBodyBytes);
```

想要实现更完善的控制，可以使用重载的变体 来指定 mandatory 标识，或者发送预设好消

息属性的消息

```
channel.basicPublish(exchangeName, routingKey, mandatory,
                     MessageProperties.PERSISTENT_TEXT_PLAIN,
                     messageBodyBytes);
```

以下例子 发送消息的时候会指定投递模式为2(持久化)，优先级为1 且 消息体类型(content-type) 为 text/plain。使用Builder类 去创建一个需要 指定多个属性的消息属性对象：

```
channel.basicPublish(exchangeName, routingKey,
             new AMQP.BasicProperties.Builder()
               .contentType("text/plain")
               .deliveryMode(2)
               .priority(1)
               .userId("bob")
               .build(),
               messageBodyBytes);
```

以下是发布带有自定义headers消息的例子：

```
Map<String, Object> headers = new HashMap<String, Object>();
headers.put("latitude",  51.5252949);
headers.put("longitude", -0.0905493);

channel.basicPublish(exchangeName, routingKey,
             new AMQP.BasicProperties.Builder()
               .headers(headers)
               .build(),
               messageBodyBytes);
```

下面是发布一条 具有过期时间属性 的消息

```
channel.basicPublish(exchangeName, routingKey,
             new AMQP.BasicProperties.Builder()
               .expiration("60000")
               .build(),
               messageBodyBytes);
```

通道和并发的注意事项(线程安全)

依经验而言，应该尽量 避免 线程间 共享 channel。应用应该 尽可能 每个线程使用单独的通道，而不是将 通道 共享给 多个线程。

虽然可以安全地并发 调用channel 上的某些操作，但是有些操作 不能并发调用，会导致 错误的帧交错在网络上，或造成重复确认等问题。

在共享的通道上并发执行publish，会导致错误的帧交错在网络上，触发 channel 级别的协议异常 并导致 connection被 broker 直接关闭。 因此，需要在应用程序代码中 进行显示同步。线程间 共享channel 也会干扰 publish-ack。最好能完全避免 在共享的通道上 进行并发 发布。

。。那就是 读-写锁。 不，感觉 读同一个channel也会出问题。。不，哪有并发读，那个consumer 是一个 lambda 来处理的。

也可以通过 channel pool 的方式 来避免 在共享通道上 并发 发布消息： 一旦一个线程使用完了 某个通道，就将通道归还到 池中，使得 channel 可以被其他线程 再次使用。channel pool 可以视为一个特殊的 同步解决方案。 建议使用现有的 pool，而不是自己实现。例如 开箱即用的 Spring AMQP。

channel 是 吃资源的，而且 大多数应用场景下， 同一个jvm进程 很少会开放 数百个channel。

一个需要避免的 经典的 反模式 就是 为 每个发布的消息 开放单独的 channel， channel应该是 长时间存活的，并且 开放一个channel 是一个 网络往返的过程，所以这种模式 是低效的。

一个线程用于消费，另一个线程在 共享通道上 推送 是安全的。

服务推送投递 是以并发的方式 分发的，并且 能确保 每个channel 顺序是固定的。 分发机制 在每个 连接中 使用一个 java.util.concurrent.ExecutorService。 使用ConnectionFactory#setSharedExecutor setter 的ConnectionFactory生成的所有连接都可以共享一个自定义的executor。

当使用 手动ack 时，需要考虑到的是 线程完成的 确认动作。 这和 线程收取 投递(如Consumer#handleDelivery 将 交付处理委派给 其他线程)不同， 确认操作将 multiple 这个参数 设置成 true 是不安全的，可以会导致 2次 确认， 还会触发 channel 级别异常 并且 关闭 channel。 在 同一时间 单独 确认一条 独立的消息 是没有问题的。

通过订阅来接收消息("推送接口")
接收消息 最高效 的方式 是使用 Consumer接口 设置 订阅。消息在到达时被自动投递到其中，而不是显式地 去 请求。
当调用 Consumers 相关的接口方法时，单个订阅 始终由 其消费者标签引用。consumerTag 可以由 客户端 或 服务器 来生成，用于 消费者的身份识别。想让RabbitMQ 生成一个 节点范围内的唯一标签，可以使用 不含 消费者标签属性 的 Channel#basicConsume 重载，或者传递一个 空字符串 作为 consumer tag，然后 使用 Channel#basicConsume 返回的值。消费者标签 可以用于 清除消费者。

不同的消费者实例必须持有不同的消费者标签。非常不建议在 同一个 连接上 出现重复的消费者标签，这会导致 自动连接覆盖 问题，并在 监控消费者时 混淆监控数据。

实现Consumer 的最最简单方式 是 子类化 DefaultConsumer。此子类的 实例化对象 可以被当做 basicConsume 调用时的 参数进行传递，用于设置订阅：

```
boolean autoAck = false;
channel.basicConsume(queueName, autoAck, "myConsumerTag",
     new DefaultConsumer(channel) {
         @Override
         public void handleDelivery(String consumerTag,
                                    Envelope envelope,
                                    AMQP.BasicProperties properties,
                                    byte[] body)
             throws IOException
```

```
        {
            String routingKey = envelope.getRoutingKey();
            String contentType = properties.getContentType();
            long deliveryTag = envelope.getDeliveryTag();
            // (process the message components here ...)
            channel.basicAck(deliveryTag, false);
        }
    });
```

上面设置了 autoAck=false，所以需要手动对 投递到 Consumer 的消息进行确认。

更复杂的消费者 需要去重写其他方法。 需要说明的是：当channel 和 connection 关闭时， handleShutdownSignal 会被调用，handleConsumeOk 会在调用其他 Consumer 回调 之前被传递给 消费者标签。

Consumers 同样可以通过实现 handleCancelOk 和 handleCancel 方法 来分别 被 告知 是通过 显式还是隐式方法进行取消。

你可以通过 Channel.basicCancel 显式地取消 一个指定的 Consumer。

```
channel.basicCancel(consumerTag);
```

和发布者一样，这里也要考虑 消费者的 并发安全性。
消费者的回调 的 调度 是在一个独立的线程池中完成的，这个线程池跟通道实例化的那个池是分开的。这表示 Consumers 可以安全地 调用类似于 Channel#queueDeclare 和 Channel#basicCancel 这种连接 和通道的 阻塞方法。

每个channel 都有自己的 调度线程。对于大多数 常见的 一个channel 一个consumer 的场景下，这意味着 消费者 之间不会相互影响。需要注意，如果一个通道里 有多个消费者，长时间运行的消费者 会阻挡 channel中其他消费者的 回调方法的 调度。

检索单条消息("拉取接口")
按需检索单条消息也是可以的(pull API 又名 polling)。这种消费方法的效率是极低的，比如，它使用的是 轮询的方式， 即使大多数 请求结果尚未生成，应用也会 重复地去请求结果。因此，这种方法 是 强烈 不 建议 使用的。

使用 Channel.basicGet 来进行消息的 拉取， 返回值 是包含有 头信息 和 消息体 的 GetResponse 对象实例。

```
boolean autoAck = false;
GetResponse response = channel.basicGet(queueName, autoAck);
if (response == null) {
    // No message retrieved.
} else {
    AMQP.BasicProperties props = response.getProps();
    byte[] body = response.getBody();
    long deliveryTag = response.getEnvelope().getDeliveryTag();
    // ...

// 一般在成功处理后，进行 ack。
```

```
    channel.basicAck(method.deliveryTag, false); // acknowledge receipt of the
    message
}
```

处理无法路由的消息
如果发布的消息 设置了 mandatory 标识，但是没有被 路由成功，代理 会将其返回给 发送的客户端(通过 AMQP.Bacis.Return 命令)
客户端可以通过实现 ReturnListener 接口，并调用 Channel.addReturnListener 来收到此类通知，如果客户端没有为channel配置 return listener，那么 相应的消息会被 丢弃掉。

```
channel.addReturnListener(new ReturnListener() {
    public void handleReturn(int replyCode,
                             String replyText,
                             String exchange,
                             String routingKey,
                             AMQP.BasicProperties properties,
                             byte[] body)
    throws IOException {
        ...
    }
});
```

例如，客户端发布了一条带有 mandatory 标识的消息，此消息设置了 交换机类型 为 direct，但是 交换机 没有绑定到 队列尚，此时 return listener 就会被调用。

关闭协议
客户端关闭进程概览
AMQP 0-9-1 connection和channel 使用相同的 通用方法 来管理网络故障，内部故障，和显式本地关闭。
AMQP 091 connection 和 channel 有以下的生命周期状态：

- 打开：对象可以使用了
- 正在关闭：已经明确通知对象在本地进行关闭，已经向所有支持的 底层 对象发出了关闭请求，并且正在等待 其 关闭过程 完成。
- 已关闭：对象已经 接收到 所有底层 对方 发出的 关闭完成的 通知，然后自己也完成了 关闭操作。

这些对象只管 完成关闭状态，而不关心 造成关闭的原因是什么。像应用请求、客户端内部库错误、远程网络请求或者网络错误一概不管。

连接和通道对象会处理如下所示的跟关闭有关的方法：

- addShutdownListener(ShutdownListener listener) 和 removeShutdownListener(ShutdownListener listener)用来管理监听器，当对象转换为关闭状态时触发。需要注意的是，给一个已经关闭的对象添加关闭监听器会立即出发监听行为。
- getCloseReason()用来获取对象关闭的原因
- isOpen()在测试对象开启状态时很有用
- close(int closeCode, String closeMessage)用来显式地通知对象执行关闭操作

```
import com.rabbitmq.client.ShutdownSignalException;
import com.rabbitmq.client.ShutdownListener;

connection.addShutdownListener(new ShutdownListener() {
    public void shutdownCompleted(ShutdownSignalException cause)
    {
        ...
    }
});
```

有关关闭情况的信息
可以通过显示调用 getCloseReason() 方法 或通过 使用带有 cause 参数的 ShutdownListener类的 service(ShutdownSignalException cause) 来获得 ShutdownSignalException，其中包含有关关闭原因的所有可用信息。

ShutdownSignalException类提供了用于分析关闭原因的方法。通过调用isHardError()方法，我们可以知道是不是因为连接或者通道错误导致，getReason()则会以返回AMQP方法的方式提供关闭的有关信息，包括AMQP.Channel.Close或者AMQP.Connection.Close（如果是客户端库引起的异常，比如网络通讯失败则会返回null，这种情况可以通过getCause()来获取异常）

```
public void shutdownCompleted(ShutdownSignalException cause)
{
  if (cause.isHardError())
  {
    Connection conn = (Connection)cause.getReference();
    if (!cause.isInitiatedByApplication())
    {
      Method reason = cause.getReason();
      ...
    }
    ...
  } else {
    Channel ch = (Channel)cause.getReference();
    ...
  }
}
```

原子性 和 isOpen()方法的 使用
由于通道 和 连接 的 isOpen() 的返回值 依赖于 关闭原因 是否存在，所以在 生产中，不建议使用这个方法。
下面的代码 对 竞争条件的 可能性进行了说明

```
public void brokenMethod(Channel channel)
{
    if (channel.isOpen())
    {
```

```
        // The following code depends on the channel being in open state.
        // However there is a possibility of the change in the channel state
        // between isOpen() and basicQos(1) call
        ...
        channel.basicQos(1);
    }
}
```

相反，我们应该忽略类似的检查，简单地尝试 自己所需要执行的 动作即可。如果代码执行过程中 channel 或 connection 关闭了， 则抛出 ShutdownSignalException 来表示 对象是 无效的。除此之外，我们还需要 捕获 由于 代理 意外关闭连接 造成的 SocketException 和 代理发起关闭请求 而造成的 ShutdownSignalException 所引发的 IOException。

```
public void validMethod(Channel channel)
{
    try {
        ...
        channel.basicQos(1);
    } catch (ShutdownSignalException sse) {
        // possibly check if channel was closed
        // by the time we started action and reasons for
        // closing it
        ...
    } catch (IOException ioe) {
        // check why connection was closed
        ...
    }
}
```

高级connection选项
消费者操作线程池
默认情况下，消费者线程 会通过一个新的 ExecutorService 线程池分配。如果需要更大的控制权，可以使用 newConnection() 去应用 ExecutorService 。
下面应用了一个 比默认分配的 更大的线程池

```
ExecutorService es = Executors.newFixedThreadPool(20);
Connection conn = factory.newConnection(es);
```

connection关闭时， 默认提供的 ExecutorService 也会执行 shutdown()， 但是 用户提供的 ExecutorService 不会 执行 shutdown()， 自定义线程池的客户端 必须 自己确保 线程池的 关闭(即调用 shutdown方法)，否则 线程池会影响JVM的 中止。
。。自己的 demo 是不是这个原因？ 但是 自己的demo 是 默认的 线程池。

相同的 ExecutorService 可能被多个 connection 共享，或者被 重复使用， 但是无论如何， 当它关闭后，不可以再被使用。

应该在 有证据表明 处理 消费回调 存在 严重瓶颈 是 才 考虑这个 功能。

主机列表的使用
把一个 Address 数组 传给 newConnection() 是可以的。Address 是 com.rabbitmq.client 中的 包含 host 和 port 的 简单的 便捷类。

```
Address[] addrArr = new Address[]{ new Address(hostname1, portnumber1)
                                 , new Address(hostname2, portnumber2)};
Connection conn = factory.newConnection(addrArr);
```

这样会 先去尝试 连接 hostname1:portnumber1， 失败的话 再 尝试 hostxxx2:portxx2. 返回的 connection 是 第一次成功的 数据元素。

如果同时也提供了ExecutorService（在factory.newConnection(es, addrArr)中使用），那线程池也是对应的第一次成功连接的那个。

使用 AddressResolver 接口 实现 服务 发现
使用 AddressResolver 来 改变 连接时 的 断点 解析算法

```
Connection conn = factory.newConnection(addressResolver);
```

AddressResolver 接口类似于

```
public interface AddressResolver {

  List<Address> getAddresses() throws IOException;

}
```

和之前的 主机列表一样， 先尝试 返回的 第一个 Address，如果失败，则尝试 第二个，直到 成功。

如果同时也提供了ExecutorService（在factory.newConnection(es, addrArr)中使用），那线程池也是对应的第一次成功连接的那个。

AddressResolver 是 实现 自定义 服务器发现逻辑的 最佳方式，在动态基础设置的状况下尤其有用。 结果 自动发现，客户端可以自动连接到 首次 启动时 尚未出现故障的 节点。姻亲(..?) 和 负载均衡 是 自定义 AddressResolver 能做的 另外2个场景。

Java 客户端 附带了 以下实现：

DnsRecordIpAddressResolver：根据给定的主机名，返回其IP地址（针对DNS服务器平台的解析）。这对简单的基于DNS的如在均衡很帮助很大。

DnsSrvRecordAddressResolver：根据给定的服务的名字，返回其所在的主机名/端口对。搜索服务基于DNS SRV请求实现。如果需要类似于HashiCorp Consul的服务注册功能的话，这也相当实用。

心跳超时
https://www.rabbitmq.com/heartbeats.html

自定义线程工厂
类似 Google App Engine (GAE) 的环境 能 限制直接将线程实例化。 想要在这种环境中 使用 RabbitMQ Java client，就需要使用 适当的方法 配置 自定义的 ThreadFactory 来实例化线程，例如 GAE 的 ThreadManager

```
import com.google.appengine.api.ThreadManager;

ConnectionFactory cf = new ConnectionFactory();
cf.setThreadFactory(ThreadManager.backgroundThreadFactory());
```

支持Java 的非阻塞 IO
Java client 4.0 带来了 对 java 非阻塞IO (NIO) 的支持， NIO的目的 不是 为了比 阻塞IO更快，而是为了更方便地 实现 简单的 资源控制(这里指的是 线程)

在默认的 阻塞IO模式下，每个connection 使用一个 线程去 网络套接字中 读取内容。在NIO 模式下，你可以控制 读取和写入的 网络套接字 的 线程的数量。

如果你的java进程 使用了 很多connection(数百个) 的情况下，可以使用 NIO。你需要使用比默认阻塞模式下更少的线程。设置合适的线程数量的 情况下，你不会损失任何性能，特别是 connection 不是特别繁忙的情况下。

NIO 需要显示 开启

```
ConnectionFactory connectionFactory = new ConnectionFactory();
connectionFactory.useNio();
```

NIO模式 可以通过 NioParams 进行配置

```
connectionFactory.setNioParams(new NioParams().setNbIoThreads(4));
```

NIO模式的默认值 是合理的，但是你 也可能需要根据自身工作负荷 来进行修改。其中一些设置包括：使用的总的IO线程数量，缓存大小，IO循环所使用的service executor,内存中写队列的参数 (将请求发送到 网络之前写入队列)

自动恢复网络故障
恢复连接

客户端 和 RabbitMQ 节点 之间的 网络连接 会发生失败， RabbitMQ的java client 支持自动 恢复 连接 和 拓扑 (队列，交换机，绑定 和 消费者)

多应用的自动恢复过程如下：

- 重连
- 恢复连接监听
- 重开通道
- 恢复通道监听
- 恢复通道的basic.qos设置，发布确认和事务设置

拓扑的恢复包括以下， 会应用到每个channel：

重新声明交换机（预定义的除外）
重新声明队列
恢复所有绑定
恢复所有消费者

java client 4.0 中，自动恢复 默认 开启 (拓扑的恢复 也是)

拓扑恢复依赖于实体的每个连接缓存（队列，交换机，绑定，消费者）。当声明一个队列的时候，此队列也会被添加到缓存中。当它被删除或者列入删除计划时（例如是一个 自动删除队列），缓存会被移除。此模型有以下限制。

使用factory.setAutomaticRecoveryEnabled(boolean)方法开启或停用自动连接恢复。以下代码片段展示了如何显式地开启自动恢复（例如针对Java客户端4.0.0之前的版本）：

```
ConnectionFactory factory = new ConnectionFactory();
factory.setUsername(userName);
factory.setPassword(password);
factory.setVirtualHost(virtualHost);
factory.setHost(hostName);
factory.setPort(portNumber);
factory.setAutomaticRecoveryEnabled(true);
// connection that will recover automatically
Connection conn = factory.newConnection();
```

如果因为异常导致恢复失败（比如RabbitMQ节点尚不可用），会在固定的时间间隔（默认5秒）进行重试。间隔可以进行配置：

```
ConnectionFactory factory = new ConnectionFactory();
// attempt recovery every 10 seconds
factory.setNetworkRecoveryInterval(10000);
```

当提供了地址列表的情况下，列表会被随机重排并逐一尝试：

```
ConnectionFactory factory = new ConnectionFactory();
Address[] addresses = {new Address("192.168.1.4"), new Address("192.168.1.5")};
factory.newConnection(addresses);
```

连接的自动恢复 何时会被触发
如果开启了 连接自动恢复，会按照以下 时间来进行触发：
连接的I/O循环中抛出了I/O异常
套接字(socket)读操作超时
检测到服务器丢失 心跳
连接的I/O循环中抛出了其他不可预期的异常
以先发生的为准。

如果client 到 RabbitMQ节点的 连接 初始化失败，连接自动恢复 不会生效。应用的开发者需要负责 重试连接，记录下失败的尝试， 实现重试的 次数限制 等。
下面是一个 基本的例子

```
ConnectionFactory factory = new ConnectionFactory();
// configure various connection settings

try {
  Connection conn = factory.newConnection();
} catch (java.net.ConnectException e) {
  Thread.sleep(5000);
  // apply retry logic
}
```

当连接被 应用通过 Connection.close 方法关闭的情况下，不会启动 连接恢复。

通道级 异常不会触发 任何 恢复， 因为这些 异常 通常是指 应用中的 语义问题。

恢复监听
可以在 可恢复的连接上 注册一个或多个 恢复监听， 当连接恢复 开启时，
ConnectionFactory#newConnection和Connection#createChannel返回的连接实现了com.rabbitmq.client.Recoverable，并且提供了两个相当具有描述性名字的方法。

```
addRecoveryListener
removeRecoveryListener
```

需要将 连接 和 通道， 强转为 Revocerable 才可以使用这些方法。

对发布的影响
当连接失败时，通过 Channel.basicPublish 发布的消息 会丢失。 客户端不会 将其放入队列中 以 等待 连接恢复后 进行投递。
要确认 发布的 消息 是否已到达 RabbitMQ， 应用 需要使用 发布确认 并解决 连接失败。

拓扑恢复
拓扑的恢复 涉及到 交换机，队列，绑定 和 消费者的 恢复。 自动恢复 启用时， 拓扑恢复 也会随之 启用。
拓扑恢复可以显式关闭

```
ConnectionFactory factory = new ConnectionFactory();

Connection conn = factory.newConnection();
// enable automatic recovery (e.g. Java client prior 4.0.0)
factory.setAutomaticRecoveryEnabled(true);
// disable topology recovery
factory.setTopologyRecoveryEnabled(false);
```

故障检测 和 恢复的 限制

连接 的自动恢复， 有一些 局限性 和 应用开发人员 需要注意的 有意设计的 策略。
拓扑恢复 依赖于 实体的 每个 连接 缓存 (队列，交换机，绑定，消费者)。 当声明 一个 队列的时候， 此队列 也会被 添加到 缓存中。当它 被删除 或 列入删除计划时 (例如 是 一个 自动删除 队列)，缓存会被移除。这样 就可以在 不同的 channel 上 声明 和删除 实体，而不会产生 意外的结果。 这也意味着，使用自动连接 恢复的 消费者标签 (特定通道的 标识符) 在所有 通道上 必须是 唯一的。

当连接断开或丢失时，需要花费一些时间 进行检测。因此，库和 应用程序 意识到 有 连接失败之前的 一个窗口期。在这段时间内发布的 所有消息 都将 照常 进行序列化 并写入 TCP 套接字。只有通过 发布者确认 才能保证将 它们 成功交付给了 代理： 按照设计，AMQP 091的发布过程完全是 异步的。

如果在启用了 自动恢复的 连接中 检测到 套接字 或 IO操作错误，则 恢复将在 默认的 5 秒 延迟后 开始 。这种设计 假定 许多网络故障 是短暂的 并且 持续时间很短，但也不会立刻恢复。 延迟 还可以避免 在相同连接上 发生 服务器端资源 清除(例如 独占 或 自动删除队列 删除) 和打开新连接 之间的 资源竞争。

默认下，连接恢复 将以 相同的 时间间隔 进行， 直到 成功打开新连接。
通过将实现RecoveryDelayHandler的实例化对象提供给ConnectionFactory#setRecoveryDelayHandler，可以实现恢复延迟的动态化。 实现动态计算的延迟间隔应避免使用过低的值（根据经验，小于2秒就算过低了）。

当连接处于 恢复状态时， 在其通道上 尝试进行的 任何发布 都将被拒绝，但也有例外。客户端当前不对 此类 output消息 进行任何 缓冲。 跟踪此类信息 并在 恢复成功后 重新发布它们 是 开发者的责任。

当发生 通道级异常 而导致 通道关闭时，连接恢复不会生效。 此类异常 通常表示 是 应用程序级别的问题。库无法在这种情况下 采取适合的 措施。

如果 通过 显式关闭 或 由于 通道级别的异常 使得 通道关闭。即使 启动了 连接恢复，也不会 恢复 已关闭的 通道。

手动确认 和 自动恢复
当使用手动确认的情况下，可能会发生 连接 在消息投递成功 但并未进行确认的 空挡中失效的情况。在连接恢复后，RabbitMQ会在通道里重置投递标签。

这意味着旧的投递标签的basic.ack, basic.nack, 和 basic.reject 会导致通道异常发生。为了避免这种情况，RabbitMQ的Java客户端会保持对投递标签的追踪和更新，以使它们在恢复过程中单调增长。

之后，Channel.basicAck, Channel.basicNack, 和 Channel.basicReject会将调整后的传递标签转换为RabbitMQ使用的传递标签。

带有过时的投递标签的确认将不会发送。使用手动确认和自动恢复的应用必须能够对重新投递的消息进行处理。

通道的生命周期和拓扑恢复
连接自动恢复对应用程序开发人员来说应尽可能透明，这就是为什么即使好几个连接失效，然后在后台恢复的情况下，Channel实例任然会保持相同的原因。 从技术上讲，启用自动恢复时，通道实例充当代理或装饰器：它们将AMQP业务委派给实际的AMQP通道实现，并围绕它实施一些恢复机制。 这就是为什么您不应该在通道完成了一些资源创建（队列，交换，绑定）之后对其进行关闭，这会导致稍后的扑恢复失败。 在应用程序的整个生命周期中都应该保持通道的打开状态。

未处理的异常
跟连接、通道、恢复和消费者生命周期有关的未处理的异常会委托给“异常处理”。“异常处理”是对ExceptionHandler接口的一个实现。默认情况下，会使用DefaultExceptionHandler实例。它会把异常的细节打印到标准输出中。

也可以用ConnectionFactory#setExceptionHandler来覆盖默认异常处理。这会应用到所有通过工厂创建的连接中。
```
ConnectionFactory factory = new ConnectionFactory();
cf.setExceptionHandler(customHandler);
```
异常处理应该将异常记录到日志当中。

指标和监控
客户端会收集活动的连接的运行时指标（例如发布消息的数量）。指标收集是需要在ConnectionFactory级别使用setMetricsCollector(metricsCollector)方法进行配置的可选功能。此方法需要的MetricsCollector实例会在客户端代码中的多处用到。

4.3版本的客户端开始支持 Micrometer 和 Dropwizard Metrics ，开箱即用。

以下是收集的指标：
- 开启的连接数量
- 开启的通道数量
- 发布的消息数量
- 消费的消息数量
- 确认的消息数量
- 拒绝的消息数量

Micrometer 和 Dropwizard Metrics 都提供了与消息指标相关的计数器，也提供了平均速率，最后5分钟速率等。他们也支持用于监控和报告的通用工具（如 JMX, Graphite, Ganglia, Datadog等）。

启用指标收集时，开发人员应牢记一些注意事项。
- 要使用Micrometer 或者 Dropwizard Metrics 的话，别忘了添加相关依赖（ (在Maven, Gradle, 或者 JAR 文件里)）到JVM classpath中。
- 指标的收集是可以扩展的。推荐为特定目的实现自定义的MetricsCollector。
- 虽然MetricsCollector是在ConnectionFactory层定义的，但是也可以在不同的实例中共享。
- 指标的收集不支持事务。举例来说，如果一个确认（acknowledgment）通过事务发送，然后事务回滚了，那这个确认就已经被客户端指标（很显然不是通过代理）累计了。需

要注意的是，确认（acknowledgment）确实已经发送到代理了，然后又被事务回滚给清除了，所以客户端指标对于确认的处理是没问题的。总之，不要把客户端指标用作关键的业务逻辑，因为不保证它们完全准确无误。它们存在的目的在于简单的解释系统的运行情况并且让操作更具效率。

Micrometer的支持

```
ConnectionFactory connectionFactory = new ConnectionFactory();
MicrometerMetricsCollector metrics = new MicrometerMetricsCollector();
connectionFactory.setMetricsCollector(metrics);
...
metrics.getPublishedMessages(); // get Micrometer's Counter object
```

Micrometer支持 多种报告后台：Netflix Atlas, Prometheus, Datadog, Influx, JMX, 等。
通常情况下会将MeterRegistry的实例传给MicrometerMetricsCollector，这里是使用JMX的示例：

```
JmxMeterRegistry registry = new JmxMeterRegistry();
MicrometerMetricsCollector metrics = new MicrometerMetricsCollector(registry);
ConnectionFactory connectionFactory = new ConnectionFactory();
connectionFactory.setMetricsCollector(metrics);
```

Dropwizard Metrics的支持

```
ConnectionFactory connectionFactory = new ConnectionFactory();
StandardMetricsCollector metrics = new StandardMetricsCollector();
connectionFactory.setMetricsCollector(metrics);
...
metrics.getPublishedMessages(); // get Metrics' Meter object
```

Dropwizard Metrics支持多种报告后台: console, JMX, HTTP, Graphite, Ganglia, 等.

通常你可以将MetricsRegistry实例传给StandardMetricsCollector，以下是关于JMX的示例：

```
MetricRegistry registry = new MetricRegistry();
StandardMetricsCollector metrics = new StandardMetricsCollector(registry);

ConnectionFactory connectionFactory = new ConnectionFactory();
connectionFactory.setMetricsCollector(metrics);

JmxReporter reporter = JmxReporter
  .forRegistry(registry)
  .inDomain("com.rabbitmq.client.jmx")
  .build();
reporter.start();
```

Google App Engine上的RabbitMQ Java客户端
要在Google App Engine (GAE)上使用RabbitMQ Java客户端的话，需要使用GAE's ThreadManager (see above)这个自定义的线程工具去实例化线程。另外需要设置一个比较低的心跳间隔（4-5秒）来避免GAE上的InputStream读取超时过低。

```
ConnectionFactory factory = new ConnectionFactory();
cf.setRequestedHeartbeat(5);
```

注意事项和限制
为了实现拓扑的自动恢复，RabbitMQ Java客户端维护了一个用于声明队列、交换机和绑定的缓存。缓存是按连接来对应的。欧谢RabbitMQ的功能会导致客户端无法观察到拓扑功能的变更，例如当队列因TTL被删除的情况。大多数情况下，RabbitMQ Java客户端会尝试让缓存实体无效：

When a queue is deleted.
当队列被删除的时候。
When an exchange is deleted.
当交换机被删除的时候。
When a binding is deleted.
当绑定被删除的时候。
When a consumer is cancelled on an auto-deleted queue.
当消费者因队列的自动删除动作而被清理掉的时候。
When a queue or exchange is unbound from an auto-deleted exchange.
当交换机或队列从自动删的的交换机上解绑的时候。

但是，客户端无法跟踪单个连接以外的拓扑变化。 依赖于自动删除队列或交换机以及队列TTL（请注意：不是消息TTL！）和使用连接自动恢复的应用应该显式删除已知的未被使用或已被删除的实体，以清除客户端拓扑缓存。 Channel＃queueDelete，Channel＃exchangeDelete，Channel＃queueUnbind和Channel＃exchangeUnbind在RabbitMQ 3.3.x中是幂等的（删除不存在的内容不会导致异常）这个特性有助于实现这一操作。

远程过程调用-RPC (请求/回复)模式: 示例
为了方便编写程序，Java客户端提供了使用一个临时回复队列来实现的RpcClient类，这样就通过AMQP 0-9-1 实现了简单的 RPC-风格通讯。

此类没有在RPC属性和返回值方面新增任何特殊格式。它只是简单的实现了附带路由键发送消息到给定的交换机，并且等待在回复队列里等待回应的机制。

```
import com.rabbitmq.client.RpcClient;

RpcClient rpc = new RpcClient(channel, exchangeName, routingKey);
```

（此类使用AMQP 0-9-1的实现细节如下：发送请求消息时，其basic.correlation_id字段设置为该RpcClient实例的唯一值，而basic.reply_to设置为回复队列的名称。）

一旦创建了此类的实例，就可以使用一下任一方法来发送RPC请求了：

```
byte[] primitiveCall(byte[] message);
String stringCall(String message)
```

```
Map mapCall(Map message)
Map mapCall(Object[] keyValuePairs)
```

primitiveCall方法将原始字节数组作为请求和响应主体进行传输。 方法stringCall是primitiveCall的一个轻量化封装，将消息正文作为默认字符编码的String实例来处理。

mapCall这种变体稍微有点复杂：他们将包含普通Java值的java.util.Map编码为AMQP 0-9-1二进制表来表示，并以相同的方式来对收到的响应进行解码。 （注意，此处可以使用哪种值类型有一些限制，详细信息请参见javadoc。）

所有编组/解组的便捷方法都使用primitiveCall作为传输机制，仅在它之上提供包装层

TLS 的支持

```
ConnectionFactory factory = new ConnectionFactory();
factory.setHost("localhost");
factory.setPort(5671);

// Only suitable for development.
// This code will not perform peer certificate chain verification and prone
// to man-in-the-middle attacks.
// See the main TLS guide to learn about peer verification and how to enable it.
factory.useSslProtocol();
```

注意，以上样例客户端默认不强制任何服务器验证（对等证书链认证），使用了”信任所有证书“的TrustManager。这在本地开发的时候很是方便，但是容易收到中间人攻击，所以 不推荐在用在生产环境中

https://www.rabbitmq.com/ssl.html
https://www.rabbitmq.com/ssl.html#java-client

OAuth 2 的支持
客户端可以通过 UAA这样的OAuth 2 服务器来进行身份认证。服务器端需要启用 OAuth 2 插件，并配置跟客户端使用同一个OAuth 2 服务器 。

获取 OAuth 2 令牌
Java客户端提供了OAuth2ClientCredentialsGrantCredentialsProvider类，用来从OAuth 2客户端凭证流获取JWT令牌。客户端会在打开连接的时候将令牌放在password字段中进行发送。然后代理会在授权之前验证JWT令牌的签名、有效性和权限，并授予对请求的虚拟主机的访问权限。

优先使用OAuth2ClientCredentialsGrantCredentialsProviderBuilder来创建OAuth2ClientCredentialsGrantCredentialsProvider实例，然后用它来配置ConnectionFactory。以下片段展示了如何为配置 OAuth 2 插件的示例设置和创建OAuth 2 credentials provider实例：

```
import com.rabbitmq.client.impl.OAuth2ClientCredentialsGrantCredentialsProvider.
        OAuth2ClientCredentialsGrantCredentialsProviderBuilder;
```

```
...
CredentialsProvider credentialsProvider =
  new OAuth2ClientCredentialsGrantCredentialsProviderBuilder()
    .tokenEndpointUri("http://localhost:8080/uaa/oauth/token/")
    .clientId("rabbit_client").clientSecret("rabbit_secret")
    .grantType("password")
    .parameter("username", "rabbit_super")
    .parameter("password", "rabbit_super")
    .build();

connectionFactory.setCredentialsProvider(credentialsProvider);
```

在生产环境中，确认令牌断电URI使用的是HTTPS，并且根据需要为HTTPS请求配置了SSLContext（用来验证和信任OAuth 2 服务器的身份）。以下代码片段使用OAuth2ClientCredentialsGrantCredentialsProviderBuilder的tls().sslContext()方法实现了上边所提及事项：

```
SSLContext sslContext = ... // create and initialise SSLContext

CredentialsProvider credentialsProvider =
  new OAuth2ClientCredentialsGrantCredentialsProviderBuilder()
    .tokenEndpointUri("http://localhost:8080/uaa/oauth/token/")
    .clientId("rabbit_client").clientSecret("rabbit_secret")
    .grantType("password")
    .parameter("username", "rabbit_super")
    .parameter("password", "rabbit_super")
    .tls()                      // configure TLS
      .sslContext(sslContext) // set SSLContext
      .builder()              // back to main configuration
    .build();
```

刷新令牌

令牌是会或过期的，代理会拒绝带有过期令牌的连接所请求的操作。可以使用CredentialsProvider#refresh()在令牌过期前使用新令牌发送请求，以防止此情况的发生。应用自己来实现是比较麻烦的，所以Java客户端提供了DefaultCredentialsRefreshService来给予一定帮助。这个工具用来追踪使用的令牌，在过期前进行刷新，并将新令牌发送给所负责的连接。

以下代码片段展示了如何创建DefaultCredentialsRefreshService实例，并且将其配置到ConnectionFactory上。

```
import com.rabbitmq.client.impl.DefaultCredentialsRefreshService.
        DefaultCredentialsRefreshServiceBuilder;
...
CredentialsRefreshService refreshService =
  new DefaultCredentialsRefreshServiceBuilder().build();
cf.setCredentialsRefreshService(refreshService);
```

DefaultCredentialsRefreshService会在令牌有效期超过80%后进行刷新，例如，如果令牌在

60分钟后过期，DefaultCredentialsRefreshService会在48分钟的时候进行刷新。这是默认的行为，更多的细节可以通过 Javadoc 了解。

==========

==========

==========