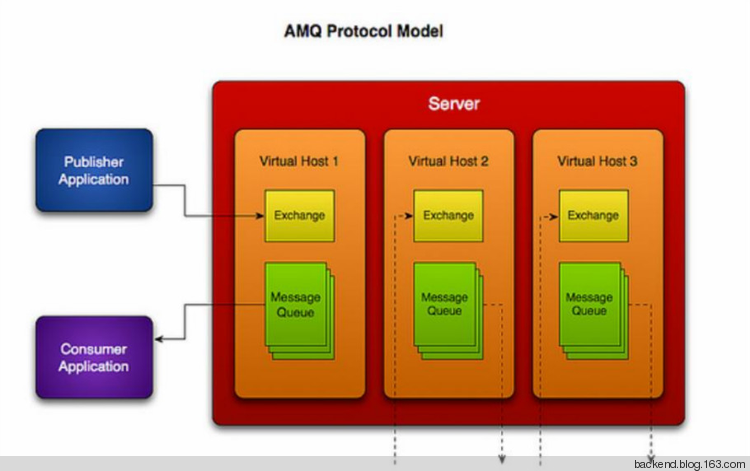
RabbitMQ主页：https://www.rabbitmq.com/

## AMQP

*AMQP协议是一个高级抽象层消息通信协议，RabbitMQ是AMQP协议的实现。它主要包括以下组件：*

**

*1.Server(broker): 接受客户端连接，实现AMQP消息队列和路由功能的进程。*

2.Virtual Host:其实是一个虚拟概念，类似于权限控制组，一个Virtual Host里面可以有若干个Exchange和Queue，但是权限控制的最小粒度是Virtual Host

3.Exchange:接受生产者发送的消息，并根据Binding规则将消息路由给服务器中的队列。ExchangeType决定了Exchange路由消息的行为，例如，在RabbitMQ中，ExchangeType有direct、Fanout和Topic三种，不同类型的Exchange路由的行为是不一样的。

4.*Message Queue：消息队列，用于存储还未被消费者消费的消息。*

*5.Message: 由Header和Body组成，Header是由生产者添加的各种属性的集合，包括Message是否被持久化、由哪个Message Queue接受、优先级是多少等。而Body是真正需要传输的APP数据。*

6.*Binding:Binding联系了Exchange与Message Queue。Exchange在与多个Message Queue发生Binding后会生成一张路由表，路由表中存储着Message Queue所需消息的限制条件即Binding Key。当Exchange收到Message时会解析其Header得到Routing Key，Exchange根据Routing Key与Exchange Type将Message路由到Message Queue。Binding Key由Consumer在Binding Exchange与Message Queue时指定，而Routing Key由Producer发送Message时指定，两者的匹配方式由Exchange Type决定。*

7.Connection:连接，对于RabbitMQ而言，其实就是一个位于客户端和Broker之间的TCP连接。

8.Channel:信道，仅仅创建了客户端到Broker之间的连接后，客户端还是不能发送消息的。需要为每一个Connection创建Channel，AMQP协议规定只有通过Channel才能执行AMQP的命令。一个Connection可以包含多个Channel。之所以需要Channel，是因为TCP连接的建立和释放都是十分昂贵的，如果一个客户端每一个线程都需要与Broker交互，如果每一个线程都建立一个TCP连接，暂且不考虑TCP连接是否浪费，就算操作系统也无法承受每秒建立如此多的TCP连接。RabbitMQ建议客户端线程之间不要共用Channel，至少要保证共用Channel的线程发送消息必须是串行的，但是建议尽量共用Connection。

9.Command:AMQP的命令，客户端通过Command完成与AMQP服务器的交互来实现自身的逻辑。例如在RabbitMQ中，客户端可以通过publish命令发送消息，txSelect开启一个事务，txCommit提交一个事务。

## Exchange类型：

direct fanout topic

### Direct

Direct Exchange - 处理路由键。需要将一个队列绑定到交换机上，要求该消息与一个特定的路由键完全匹配。这是一个完整的匹配。如果一个队列绑定到该交换机上要求路由键 “dog”，则只有被标记为“dog”的消息才被转发，不会转发dog.puppy，也不会转发dog.guard，只会转发dog。

任何发送到Direct Exchange的消息都会被转发到RouteKey中指定的Queue。

1.一般情况可以使用rabbitMQ自带的Exchange：”"(该Exchange的名字为空字符串，下文称其为default Exchange)。

2.这种模式下不需要将Exchange进行任何绑定(binding)操作

3.消息传递时需要一个“RouteKey”，可以简单的理解为要发送到的队列名字。

4.如果vhost中不存在RouteKey中指定的队列名，则该消息会被抛弃。

### Fanout

Fanout Exchange – 不处理路由键。

你只需要简单的将队列绑定到交换机上。一个发送到交换机的消息都会被转发到与该交换机绑定的所有队列上。很像子网广播，每台子网内的主机都获得了一份复制的消息。Fanout交换机转发消息是最快的。

任何发送到Fanout Exchange的消息都会被转发到与该Exchange绑定(Binding)的所有Queue上。

1.可以理解为路由表的模式

2.这种模式不需要RouteKey

3.这种模式需要提前将Exchange与Queue进行绑定，一个Exchange可以绑定多个Queue，一个Queue可以同多个Exchange进行绑定。

4.如果接受到消息的Exchange没有与任何Queue绑定，则消息会被抛弃。

### Topic

Topic Exchange – 将路由键和某模式进行匹配。此时队列需要绑定要一个模式上。符号“#”匹配一个或多个词，符号“\*”匹配不多不少一个词。因此“audit.#”能够匹配到“audit.irs.corporate”，但是“audit.\*” 只会匹配到“audit.irs”

任何发送到Topic Exchange的消息都会被转发到所有关心RouteKey中指定话题的Queue上

1.这种模式较为复杂，简单来说，就是每个队列都有其关心的主题，所有的消息都带有一个“标题”(RouteKey)，Exchange会将消息转发到所有关注主题能与

RouteKey模糊匹配的队列。

2.这种模式需要RouteKey，也许要提前绑定Exchange与Queue。

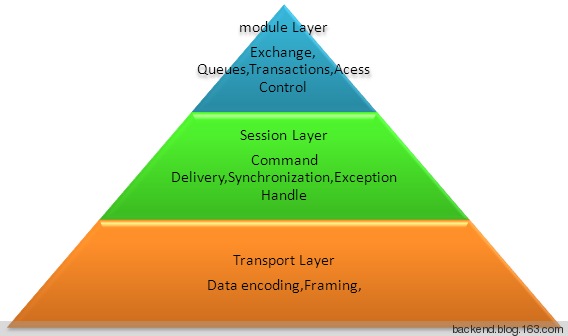
3.在进行绑定时，要提供一个该队列关心的主题，如“#.log.#”表示该队列关心所有涉及log的消息(一个RouteKey为”MQ.log.error”的消息会被转发到该队列)。

4.“#”表示0个或若干个关键字，“\*”表示一个关键字。如“log.\*”能与“log.warn”匹配，无法与“log.warn.timeout”匹配；但是“log.#”能与上述两者匹配。

5.同样，如果Exchange没有发现能够与RouteKey匹配的Queue，则会抛弃此消息。

## AMQP协议

在了解了AMQP模型以后，需要简单介绍一下AMQP的协议栈，AMQP协议本身包括三层：



*1.Module Layer，位于协议最高层，主要定义了一些供客户端调用的命令，客户端可以利用这些命令实现自己的业务逻辑，例如，客户端可以通过queue.declare声明一个队列，利用consume命令获取一个队列中的消息。*

*2.Session Layer，主要负责将客户端的命令发送给服务器，在将服务器端的应答返回给客户端，主要为客户端与服务器之间通信提供可靠性、同步机制和错误处理。*

*3.Transport Layer，主要传输二进制数据流，提供帧的处理、信道复用、错误检测和数据表示。*

*RabbitMQ使用场景*

学习RabbitMQ的使用场景，来自官方教程：https://www.rabbitmq.com/getstarted.html

场景1：单发送单接收

使用场景：简单的发送与接收，没有特别的处理。

http://images.cnitblog.com/i/434101/201408/171507104838529.png

Producer：

复制代码

import com.rabbitmq.client.ConnectionFactory;

import com.rabbitmq.client.Connection;

import com.rabbitmq.client.Channel;

public class Send {

private final static String QUEUE\_NAME = "hello";

public static void main(String[] argv) throws Exception {

ConnectionFactory factory = new ConnectionFactory();

factory.setHost("localhost");

Connection connection = factory.newConnection();

Channel channel = connection.createChannel();

channel.queueDeclare(QUEUE\_NAME, false, false, false, null);

String message = "Hello World!";

channel.basicPublish("", QUEUE\_NAME, null, message.getBytes());

System.out.println(" [x] Sent '" + message + "'");

channel.close();

connection.close();

}

}

复制代码

Consumer：

复制代码

import com.rabbitmq.client.ConnectionFactory;

import com.rabbitmq.client.Connection;

import com.rabbitmq.client.Channel;

import com.rabbitmq.client.QueueingConsumer;

public class Recv {

private final static String QUEUE\_NAME = "hello";

public static void main(String[] argv) throws Exception {

ConnectionFactory factory = new ConnectionFactory();

factory.setHost("localhost");

Connection connection = factory.newConnection();

Channel channel = connection.createChannel();

channel.queueDeclare(QUEUE\_NAME, false, false, false, null);

System.out.println(" [\*] Waiting for messages. To exit press CTRL+C");

QueueingConsumer consumer = new QueueingConsumer(channel);

channel.basicConsume(QUEUE\_NAME, true, consumer);

while (true) {

QueueingConsumer.Delivery delivery = consumer.nextDelivery();

String message = new String(delivery.getBody());

System.out.println(" [x] Received '" + message + "'");

}

}

}

复制代码

场景2：单发送多接收

使用场景：一个发送端，多个接收端，如分布式的任务派发。为了保证消息发送的可靠性，不丢失消息，使消息持久化了。同时为了防止接收端在处理消息时down掉，只有在消息处理完成后才发送ack消息。



Producer：

复制代码

import com.rabbitmq.client.ConnectionFactory;

import com.rabbitmq.client.Connection;

import com.rabbitmq.client.Channel;

import com.rabbitmq.client.MessageProperties;

public class NewTask {

private static final String TASK\_QUEUE\_NAME = "task\_queue";

public static void main(String[] argv) throws Exception {

ConnectionFactory factory = new ConnectionFactory();

factory.setHost("localhost");

Connection connection = factory.newConnection();

Channel channel = connection.createChannel();

channel.queueDeclare(TASK\_QUEUE\_NAME, true, false, false, null);

String message = getMessage(argv);

channel.basicPublish( "", TASK\_QUEUE\_NAME,

MessageProperties.PERSISTENT\_TEXT\_PLAIN,

message.getBytes());

System.out.println(" [x] Sent '" + message + "'");

channel.close();

connection.close();

}

private static String getMessage(String[] strings){

if (strings.length < 1)

return "Hello World!";

return joinStrings(strings, " ");

}

private static String joinStrings(String[] strings, String delimiter) {

int length = strings.length;

if (length == 0) return "";

StringBuilder words = new StringBuilder(strings[0]);

for (int i = 1; i < length; i++) {

words.append(delimiter).append(strings[i]);

}

return words.toString();

}

}

复制代码

发送端和场景1不同点：

1、使用“task\_queue”声明了另一个Queue，因为RabbitMQ不容许声明2个相同名称、配置不同的Queue

2、使"task\_queue"的Queue的durable的属性为true，即使消息队列durable

3、使用MessageProperties.PERSISTENT\_TEXT\_PLAIN使消息durable

When RabbitMQ quits or crashes it will forget the queues and messages unless you tell it not to. Two things are required to make sure that messages aren't lost: we need to mark both the queue and messages as durable.

Consumer：

复制代码

import com.rabbitmq.client.ConnectionFactory;

import com.rabbitmq.client.Connection;

import com.rabbitmq.client.Channel;

import com.rabbitmq.client.QueueingConsumer;

public class Worker {

private static final String TASK\_QUEUE\_NAME = "task\_queue";

public static void main(String[] argv) throws Exception {

ConnectionFactory factory = new ConnectionFactory();

factory.setHost("localhost");

Connection connection = factory.newConnection();

Channel channel = connection.createChannel();

channel.queueDeclare(TASK\_QUEUE\_NAME, true, false, false, null);

System.out.println(" [\*] Waiting for messages. To exit press CTRL+C");

channel.basicQos(1);

QueueingConsumer consumer = new QueueingConsumer(channel);

channel.basicConsume(TASK\_QUEUE\_NAME, false, consumer);

while (true) {

QueueingConsumer.Delivery delivery = consumer.nextDelivery();

String message = new String(delivery.getBody());

System.out.println(" [x] Received '" + message + "'");

doWork(message);

System.out.println(" [x] Done");

channel.basicAck(delivery.getEnvelope().getDeliveryTag(), false);

}

}

private static void doWork(String task) throws InterruptedException {

for (char ch: task.toCharArray()) {

if (ch == '.') Thread.sleep(1000);

}

}

}

复制代码

接收端和场景1不同点：

1、使用“task\_queue”声明消息队列，并使消息队列durable

2、在使用channel.basicConsume接收消息时使autoAck为false，即不自动会发ack，由channel.basicAck()在消息处理完成后发送消息。

3、使用了channel.basicQos(1)保证在接收端一个消息没有处理完时不会接收另一个消息，即接收端发送了ack后才会接收下一个消息。在这种情况下发送端会尝试把消息发送给下一个not busy的接收端。

注意点：

1）It's a common mistake to miss the basicAck. It's an easy error, but the consequences are serious. Messages will be redelivered when your client quits (which may look like random redelivery), but RabbitMQ will eat more and more memory as it won't be able to release any unacked messages.

2）Note on message persistence

Marking messages as persistent doesn't fully guarantee that a message won't be lost. Although it tells RabbitMQ to save the message to disk, there is still a short time window when RabbitMQ has accepted a message and hasn't saved it yet. Also, RabbitMQ doesn't do fsync(2) for every message -- it may be just saved to cache and not really written to the disk. The persistence guarantees aren't strong, but it's more than enough for our simple task queue. If you need a stronger guarantee you can wrap the publishing code in atransaction.

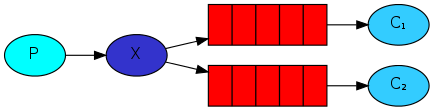
3）Note about queue size

If all the workers are busy, your queue can fill up. You will want to keep an eye on that, and maybe add more workers, or have some other strategy.

4）RabbitMQ allows you to set Time To Live for both messages and queues. https://www.rabbitmq.com/ttl.html

场景3：Publish/Subscribe

使用场景：发布、订阅模式，发送端发送广播消息，多个接收端接收。



Producer：

复制代码

import com.rabbitmq.client.ConnectionFactory;

import com.rabbitmq.client.Connection;

import com.rabbitmq.client.Channel;

public class EmitLog {

private static final String EXCHANGE\_NAME = "logs";

public static void main(String[] argv) throws Exception {

ConnectionFactory factory = new ConnectionFactory();

factory.setHost("localhost");

Connection connection = factory.newConnection();

Channel channel = connection.createChannel();

channel.exchangeDeclare(EXCHANGE\_NAME, "fanout");

String message = getMessage(argv);

channel.basicPublish(EXCHANGE\_NAME, "", null, message.getBytes());

System.out.println(" [x] Sent '" + message + "'");

channel.close();

connection.close();

}

private static String getMessage(String[] strings){

if (strings.length < 1)

return "info: Hello World!";

return joinStrings(strings, " ");

}

private static String joinStrings(String[] strings, String delimiter) {

int length = strings.length;

if (length == 0) return "";

StringBuilder words = new StringBuilder(strings[0]);

for (int i = 1; i < length; i++) {

words.append(delimiter).append(strings[i]);

}

return words.toString();

}

}

复制代码

发送端：

发送消息到一个名为“logs”的exchange上，使用“fanout”方式发送，即广播消息，不需要使用queue，发送端不需要关心谁接收。

Consumer：

复制代码

import com.rabbitmq.client.ConnectionFactory;

import com.rabbitmq.client.Connection;

import com.rabbitmq.client.Channel;

import com.rabbitmq.client.QueueingConsumer;

public class ReceiveLogs {

private static final String EXCHANGE\_NAME = "logs";

public static void main(String[] argv) throws Exception {

ConnectionFactory factory = new ConnectionFactory();

factory.setHost("localhost");

Connection connection = factory.newConnection();

Channel channel = connection.createChannel();

channel.exchangeDeclare(EXCHANGE\_NAME, "fanout");

String queueName = channel.queueDeclare().getQueue();

channel.queueBind(queueName, EXCHANGE\_NAME, "");

System.out.println(" [\*] Waiting for messages. To exit press CTRL+C");

QueueingConsumer consumer = new QueueingConsumer(channel);

channel.basicConsume(queueName, true, consumer);

while (true) {

QueueingConsumer.Delivery delivery = consumer.nextDelivery();

String message = new String(delivery.getBody());

System.out.println(" [x] Received '" + message + "'");

}

}

}

复制代码

接收端：

1、声明名为“logs”的exchange的，方式为"fanout"，和发送端一样。

2、channel.queueDeclare().getQueue();该语句得到一个随机名称的Queue，该queue的类型为non-durable、exclusive、auto-delete的，将该queue绑定到上面的exchange上接收消息。

3、注意binding queue的时候，channel.queueBind()的第三个参数Routing key为空，即所有的消息都接收。如果这个值不为空，在exchange type为“fanout”方式下该值被忽略！

场景4：Routing (按路线发送接收)

使用场景：发送端按routing key发送消息，不同的接收端按不同的routing key接收消息。



Producer：

复制代码

import com.rabbitmq.client.ConnectionFactory;

import com.rabbitmq.client.Connection;

import com.rabbitmq.client.Channel;

public class EmitLogDirect {

private static final String EXCHANGE\_NAME = "direct\_logs";

public static void main(String[] argv) throws Exception {

ConnectionFactory factory = new ConnectionFactory();

factory.setHost("localhost");

Connection connection = factory.newConnection();

Channel channel = connection.createChannel();

channel.exchangeDeclare(EXCHANGE\_NAME, "direct");

String severity = getSeverity(argv);

String message = getMessage(argv);

channel.basicPublish(EXCHANGE\_NAME, severity, null, message.getBytes());

System.out.println(" [x] Sent '" + severity + "':'" + message + "'");

channel.close();

connection.close();

}

private static String getSeverity(String[] strings){

if (strings.length < 1)

return "info";

return strings[0];

}

private static String getMessage(String[] strings){

if (strings.length < 2)

return "Hello World!";

return joinStrings(strings, " ", 1);

}

private static String joinStrings(String[] strings, String delimiter, int startIndex) {

int length = strings.length;

if (length == 0 ) return "";

if (length < startIndex ) return "";

StringBuilder words = new StringBuilder(strings[startIndex]);

for (int i = startIndex + 1; i < length; i++) {

words.append(delimiter).append(strings[i]);

}

return words.toString();

}

}

复制代码

发送端和场景3的区别：

1、exchange的type为direct

2、发送消息的时候加入了routing key

Consumer：

复制代码

import com.rabbitmq.client.ConnectionFactory;

import com.rabbitmq.client.Connection;

import com.rabbitmq.client.Channel;

import com.rabbitmq.client.QueueingConsumer;

public class ReceiveLogsDirect {

private static final String EXCHANGE\_NAME = "direct\_logs";

public static void main(String[] argv) throws Exception {

ConnectionFactory factory = new ConnectionFactory();

factory.setHost("localhost");

Connection connection = factory.newConnection();

Channel channel = connection.createChannel();

channel.exchangeDeclare(EXCHANGE\_NAME, "direct");

String queueName = channel.queueDeclare().getQueue();

if (argv.length < 1){

System.err.println("Usage: ReceiveLogsDirect [info] [warning] [error]");

System.exit(1);

}

for(String severity : argv){

channel.queueBind(queueName, EXCHANGE\_NAME, severity);

}

System.out.println(" [\*] Waiting for messages. To exit press CTRL+C");

QueueingConsumer consumer = new QueueingConsumer(channel);

channel.basicConsume(queueName, true, consumer);

while (true) {

QueueingConsumer.Delivery delivery = consumer.nextDelivery();

String message = new String(delivery.getBody());

String routingKey = delivery.getEnvelope().getRoutingKey();

System.out.println(" [x] Received '" + routingKey + "':'" + message + "'");

}

}

}

复制代码

接收端和场景3的区别：

在绑定queue和exchange的时候使用了routing key，即从该exchange上只接收routing key指定的消息。

场景5：Topics (按topic发送接收)

使用场景：发送端不只按固定的routing key发送消息，而是按字符串“匹配”发送，接收端同样如此。



Producer：

复制代码

import com.rabbitmq.client.ConnectionFactory;

import com.rabbitmq.client.Connection;

import com.rabbitmq.client.Channel;

public class EmitLogTopic {

private static final String EXCHANGE\_NAME = "topic\_logs";

public static void main(String[] argv) {

Connection connection = null;

Channel channel = null;

try {

ConnectionFactory factory = new ConnectionFactory();

factory.setHost("localhost");

connection = factory.newConnection();

channel = connection.createChannel();

channel.exchangeDeclare(EXCHANGE\_NAME, "topic");

String routingKey = getRouting(argv);

String message = getMessage(argv);

channel.basicPublish(EXCHANGE\_NAME, routingKey, null, message.getBytes());

System.out.println(" [x] Sent '" + routingKey + "':'" + message + "'");

}

catch (Exception e) {

e.printStackTrace();

}

finally {

if (connection != null) {

try {

connection.close();

}

catch (Exception ignore) {}

}

}

}

private static String getRouting(String[] strings){

if (strings.length < 1)

return "anonymous.info";

return strings[0];

}

private static String getMessage(String[] strings){

if (strings.length < 2)

return "Hello World!";

return joinStrings(strings, " ", 1);

}

private static String joinStrings(String[] strings, String delimiter, int startIndex) {

int length = strings.length;

if (length == 0 ) return "";

if (length < startIndex ) return "";

StringBuilder words = new StringBuilder(strings[startIndex]);

for (int i = startIndex + 1; i < length; i++) {

words.append(delimiter).append(strings[i]);

}

return words.toString();

}

}

复制代码

发送端和场景4的区别：

1、exchange的type为topic

2、发送消息的routing key不是固定的单词，而是匹配字符串，如"\*.lu.#"，\*匹配一个单词，#匹配0个或多个单词。

Consumer：

复制代码

import com.rabbitmq.client.ConnectionFactory;

import com.rabbitmq.client.Connection;

import com.rabbitmq.client.Channel;

import com.rabbitmq.client.QueueingConsumer;

public class ReceiveLogsTopic {

private static final String EXCHANGE\_NAME = "topic\_logs";

public static void main(String[] argv) {

Connection connection = null;

Channel channel = null;

try {

ConnectionFactory factory = new ConnectionFactory();

factory.setHost("localhost");

connection = factory.newConnection();

channel = connection.createChannel();

channel.exchangeDeclare(EXCHANGE\_NAME, "topic");

String queueName = channel.queueDeclare().getQueue();

if (argv.length < 1){

System.err.println("Usage: ReceiveLogsTopic [binding\_key]...");

System.exit(1);

}

for(String bindingKey : argv){

channel.queueBind(queueName, EXCHANGE\_NAME, bindingKey);

}

System.out.println(" [\*] Waiting for messages. To exit press CTRL+C");

QueueingConsumer consumer = new QueueingConsumer(channel);

channel.basicConsume(queueName, true, consumer);

while (true) {

QueueingConsumer.Delivery delivery = consumer.nextDelivery();

String message = new String(delivery.getBody());

String routingKey = delivery.getEnvelope().getRoutingKey();

System.out.println(" [x] Received '" + routingKey + "':'" + message + "'");

}

}

catch (Exception e) {

e.printStackTrace();

}

finally {

if (connection != null) {

try {

connection.close();

}

catch (Exception ignore) {}

}

}

}

}

复制代码

接收端和场景4的区别：

1、exchange的type为topic

2、接收消息的routing key不是固定的单词，而是匹配字符串。

注意点：

Topic exchange

Topic exchange is powerful and can behave like other exchanges. When a queue is bound with "#" (hash) binding key - it will receive all the messages, regardless of the routing key - like in fanout exchange. When special characters "\*" (star) and "#" (hash) aren't used in bindings, the topic exchange will behave just like a direct one.

About queue：

1 queue-name

The queue name MAY be empty, in which case the server MUST create a new queue with a unique generated name and return this to the client in the Declare-Ok method.

2 passive

If set, the server will reply with Declare-Ok if the queue already exists with the same name, and raise an error if not.

3 durable

If set when creating a new queue, the queue will be marked as durable. Durable queues remain active when a server restarts. Non-durable queues (transient queues) are purged if/when a server restarts. Note that durable queues do not necessarily hold persistent messages, although it does not make sense to send persistent messages to a transient queue.

The server MUST recreate the durable queue after a restart.

The server MUST support both durable and transient queues.

4 exclusive

Exclusive queues may only be accessed by the current connection, and are deleted when that connection closes. Passive declaration of an exclusive queue by other connections are not allowed.

The server MUST support both exclusive (private) and non-exclusive (shared) queues.

The client MAY NOT attempt to use a queue that was declared as exclusive by another still-open connection. Error code: [resource-locked](http://www.rabbitmq.com/amqp-0-9-1-reference.html#constant.resource-locked)

5 auto-delete

If set, the queue is deleted when all consumers have finished using it. The last consumer can be cancelled either explicitly or because its channel is closed. If there was no consumer ever on the queue, it won't be deleted. Applications can explicitly delete auto-delete queues using the Delete method as normal.

The server MUST ignore the auto-delete field if the queue already exists.

About exchange：

1 passive

If set, the server will reply with Declare-Ok if the exchange already exists with the same name, and raise an error if not.

2 durable

If set when creating a new exchange, the exchange will be marked as durable. Durable exchanges remain active when a server restarts. Non-durable exchanges (transient exchanges) are purged if/when a server restarts.

3 auto-delete

If set, the exchange is deleted when all queues have finished using it.

The server SHOULD allow for a reasonable delay between the point when it determines that an exchange is not being used (or no longer used), and the point when it deletes the exchange. At the least it must allow a client to create an exchange and then bind a queue to it, with a small but non-zero delay between these two actions.

The server MUST ignore the auto-delete field if the exchange already exists.

4 internal

If set, the exchange may not be used directly by publishers, but only when bound to other exchanges. Internal exchanges are used to construct wiring that is not visible to applications.

参考：

https://www.rabbitmq.com/getstarted.html

http://www.rabbitmq.com/amqp-0-9-1-reference.html#queue.declare.exclusive

https://www.rabbitmq.com/ttl.html

http://backend.blog.163.com/blog/static/202294126201322215551999/

http://www.cnblogs.com/luxiaoxun/p/3918054.html