RabbitMQ

一、MQ简介

（一）什么是MQ

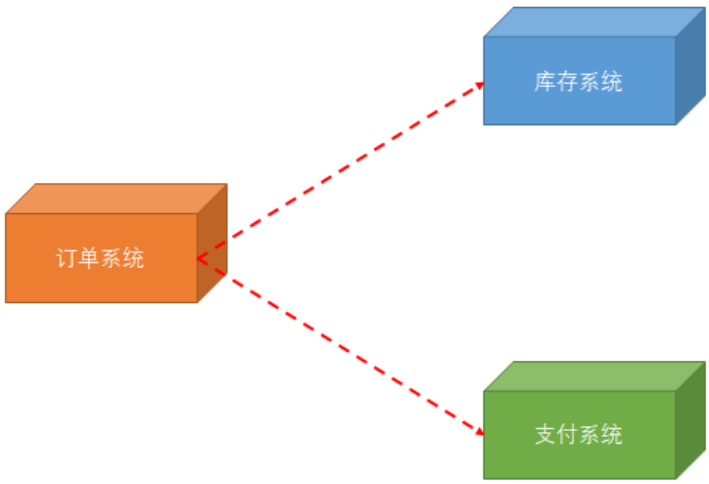
> **MQ（Message Queue）**消息队列，是基础数据结构中**“先进先出”**的一种数据结构。指把要传输的数据（消息）放在队列中，用队列机制来实现消息传递（生产者产生消息并把消息放入队列，然后由消费者去处理）。
>
> 消费者可以到指定队列拉取消息，或者订阅相应的队列，由MQ服务端给其推送消息。
>
> 一般用来解决应用解耦，异步消息，流量削峰等问题，实现高性能，高可用，可伸缩和最终一致性架构。
>
> 主要的MQ产品包括：**RabbitMQ**、ActiveMQ、RocketMQ、ZeroMQ、Kafka、IBM WebSphere等。

（二）MQ的作用?

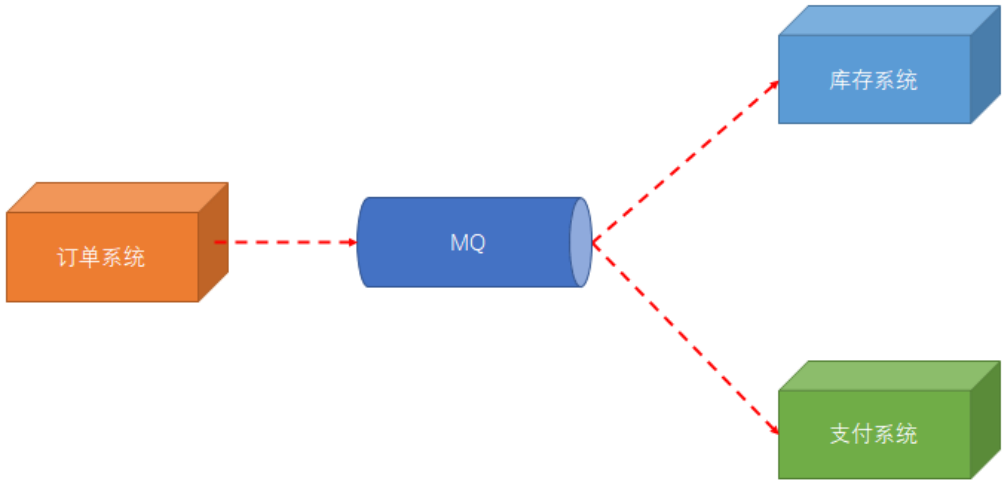
1、应用解耦

> 电商应用中，存在用户系统、订单系统、库存系统、支付系统等多个系统。用户从登录到挑选商品下单，如果所有的系统耦合在一起，其中一个系统出现问题就会导致整个流程中断，给用户带来的体验感非常差，通知后期的维护成本也非常的高。
>
> 将用户系统、订单系统、库存系统、支付系统等系统分别作为单独的系统运行，通过 MQ 消息中间在个个系统之间传递信息，极大的降低了模块之间的耦合度。比如，支付系统出现问题，并不影响用户通过订单系统下单。

未使用MQ之前：



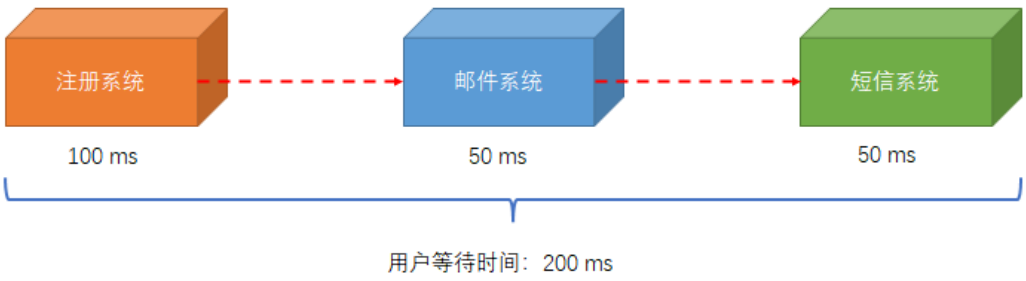
使用MQ之后：



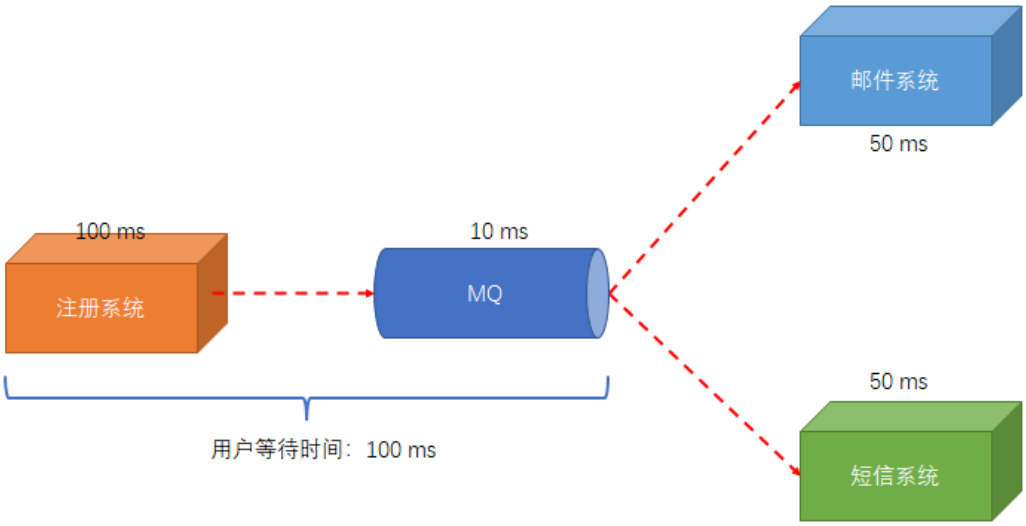
2、异步消息

> 以注册用户发送邮件或者短信为例，用户填写页面表单数据，提交至系统，系统通过处理将数据存储到 DB 中，然后还要调用发送邮件的功能代码以及发用短信的功能代码。一套流程走完之后再给用户响应信息，提示注册成功，并发量较大的时候，用户等待时间很长，这对用户来说是非常不友好的。此时可以通过 MQ 实现异步操作。
>
> 用户注册信息存储到数据库之后即可向用户反馈注册成功，同时向 MQ 发送一条信息，邮件系统和短信系统订阅 MQ，并拉去消息进行处理。

串行处理：



并行处理：



3、流量削峰

> 双十一或者秒杀活动场景下，流量过大，会导致系统承受不了压力而宕机。可以通过加入 MQ 限制流量，进而保护系统。
>
> MQ 可以控制流量数量，多余的流量可以直接丢弃。秒杀场景下可以获取前100个流量处理，其余的流量丢弃，并给用户反馈秒杀结束信息。
>
> 还可以缓解短时间内的大流量压力。双十一大流量场景下可以提醒用户当前系统繁忙稍后再试。

（三）MQ 分类

1、ActiveMQ

> ActiveMQ 是Apache出品，最流行的，能力强劲的开源消息队列。它是一个完全支持JMS规范的的消息中间件。丰富的API，多种集群架构模式让 ActiveMQ 在业界成为老牌的消息中间件，在中小型企业颇受欢迎!
>
> 优点：单机吞吐量万级，时效性毫秒级，可用性高，基于主从架构实现高可用性，消息可靠性较低的概率丢失数据
>
> 缺点：官方社区现在对ActiveMQ5x维护越来越少，高吞吐量场景较少使用。

2、Kafka

> Kafka是 LinkedIn 开源的分布式发布-订阅消息系统，目前归属于Apache顶级项目。
>
> 优点：性能卓越，单机写入TPS约在百万条/秒，最大的优点，就是吞吐量高。时效性毫秒级可用性非常高，kafka 是分布式的，一个数据多个副本，少数机器宕机，不会丢失数据，不会导致不可用，消费者采用Pull方式获取消息消息有序通过控制能够保证所有消息被消费且仅被消费一次；有优秀的第三方 KafkaWeb 管理界面KafkaManage；在日志领域比较成熟，被多家公司和多个开源项目使用;功能支持:功能较为简单，主要支持简单的MQ功能，在大数据领域的实时计算以及日志采集被大规模使用
>
> 缺点：Kafka单机超过64个队列/分区，Load会发生明显的飙高现象，队列越多，load越高，发送消息响应时间变长，使用短轮询方式，实时性取决于轮询间隔时间，消费失败不支持重试;支持消息顺序，但是一台代理宕机后，就会产生消息乱序，社区更新较慢;

3.RdcketMQ

> RocketMQ是阿里开源的消息中间件，它是纯Java开发，具有高吐量量、高可用性、适合大规模分布式系统应用的特点。RocketMQ思路起源于Kafka，但并不是Kafka的一个Copy，它对消息的可靠传输及事务性做了优化，目前在阿里集团被广泛应用于交易、充值、流计算、消息推送、日志流式处理、binglog分发等场景。
>
> 优点：单机吞吐量十万级可用性非常高，分布式架构消息可以做到0丢失MQ功能较为完善，还是分布式的，扩展性好支持10亿级别的消息堆积，不会因为堆积导致性能下降源码是java我们可以自己阅读源码，定制自己公司的MQ
>
> 缺点：支持的客户端语言不多，目前是java及c++，其中c++不成熟;社区活跃度一般没有在MQ核心中去实现JMS等接口，有些系统要迁移需要修改大量代码

4.RabbitMQ

> RabbitMQ是使用Erlang语言开发的开源消息队列系统，基于**AMQP**协议来实现。AMQP的主要特征是面向消息、队列、路由(包括点对点和发布/订阅)、可靠性、安全。 AMQP协议更多用在企业系统内对数据一致性、稳定性和可靠性要求很高的场景，对性能和吞吐量的要求还在其次。

二、RabbitMQ

(一）RabbitMQ概念

> RabbitMQ 是一个消息中间件，主要作用是接收消息和转发消息。
>
> 可以简单的把 RabbitMQ 理解为快递站点，网购的物品（消息）送往快递点儿进行暂存，之后快递点安排快递小哥将物品送到指定的地点（转发）。

(二）RabbitMQ四大核心

- **生产者**

  产生数据发送消息的程序是生产者

- **交换机**

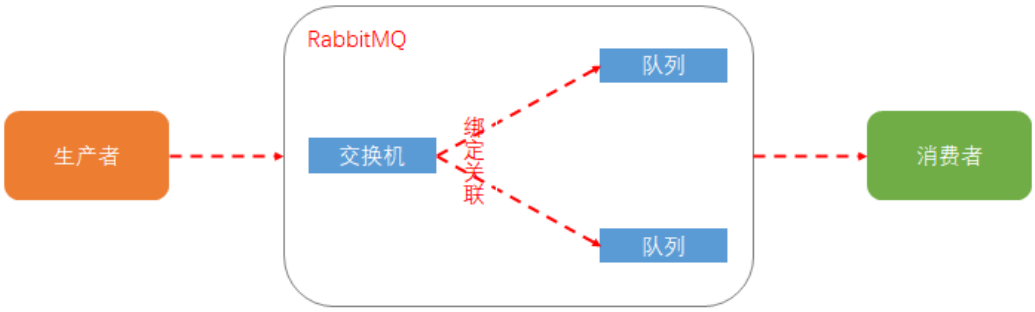
  交换机是 RabbitMQ 中非常重要的一个组件，一方面接收生产者的消息，另一方面将消息推送到队列中。交换机必须确切知道如何处理消息，是将这些消息推送到指定的队列还是多个队列，亦或者是将消息丢弃，这是由交换机的类型决定的。

- **队列**

  队列是 RabbitMQ 内部使用的数据结构，用于暂存消息，可以理解为一个缓存区。生产者可以将消息存储在队列中。消费者可以从队列中获取消息。

- **消费者**

  消费者是等待接收消息的程序。生产者、消费者、消息中间件可能不在同一个机器上。同一个应用程序可能及是生产者又是消费者。



（三）RabbitMQ 工作模式

> 官网地址：https://rabbitmq.com/getstarted.html

1. 简单模式（Hello World）
2. 工作模式（Work queues）
3. 发布订阅模式（Publish/Subscribe）
4. 路由模式（Routing）
5. 主题模式（Topics）
6. 发布确认模式（publisher Confirms）
7. RPC模式

1 "Hello World!"

The simplest thing that does *something*

- Python
- Java
- Ruby
- PHP
- C#
- JavaScript
- Go
- Elixir
- Objective-C
- Swift
- Spring AMQP

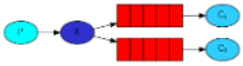
2 Work queues

Distributing tasks among workers (the competing consumers pattern)

- Python
- Java
- Ruby
- PHP
- C#
- JavaScript
- Go
- Elixir
- Objective-C
- Swift
- Spring AMQP

3 Publish/Subscribe

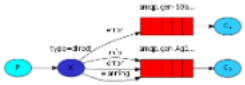
Sending messages to many consumers at once



- Python
- Java
- Ruby
- PHP
- C#
- JavaScript
- Go
- Elixir
- Objective-C
- Swift
- Spring AMQP

4 Routing

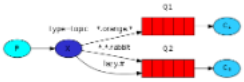
Receiving messages selectively



- Python
- Java
- Ruby
- PHP
- C#
- JavaScript
- Go
- Elixir
- Objective-C
- Swift
- Spring AMQP

5 Topics

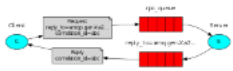
Receiving messages based on a pattern (topics)



- Python
- Java
- Ruby
- PHP
- C#
- JavaScript
- Go
- Elixir
- Objective-C
- Swift
- Spring AMQP

6 RPC

Request/reply pattern example



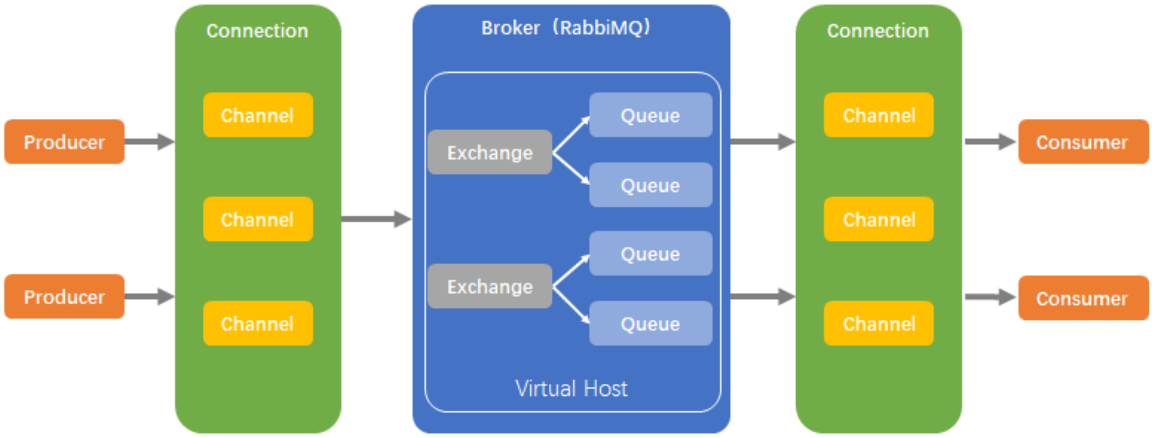
- Python
- Java
- Ruby
- PHP
- C#
- JavaScript
- Go
- Elixir
- Spring AMQP

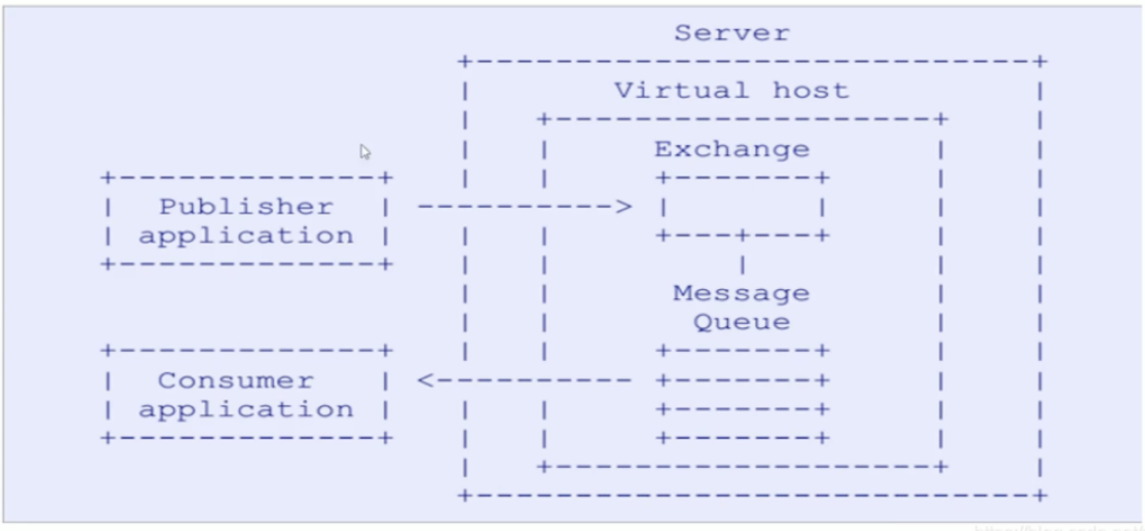
7 Publisher Confirms

Reliable publishing with publisher confirms

- Java
- C#
- PHP

（四）名词解释





- **Producer**：生产者，产生消息的应用程序；
- **Connection**：连接，publisher / consumer 和 broker 之间的 TCP 连接；
- **Channel**：信道
  - 多路复用连接中的一条独立的双向数据流通道，
  - 信道是建立在真实的 TCP 连接内的虚拟连接，AMQP 命令都是通过信道发出去的，不管是发布消息、订阅队列还是接收消息，都是通过信道完成；
  - 因为对于操作系统来说建立和销毁 TCP 都是非常昂贵的开销，所以引入了信道的概念，以复用一条 TCP 连接；
- **Broker**：表示消息队列服务器实体，接收和分发消息的应用，RabbitMQ Server 就是 Message Broker；

- **Virtual Host**：虚拟主机，表示一批交换器、消息队列和相关对象；
  - 虚拟主机是共享相同的身份认证和加密环境的独立服务器域
  - 每个 vhost 本质上就是一个 mini 版的 RabbitMQ 服务器，拥有自己的队列、交换器、绑定和权限机
  - vhost 是 AMQP 概念的基础，必须在连接时指定，RabbitMQ 默认的 vhost 是 /
- **Exchange**：交换机，用来接收生产者发送的消息并将这些消息路由给服务器中的队列
- **Binding**：绑定，用于消息队列和交换器之间进行关联。一个绑定就是基于路由键将交换器和消息队列连接起来的路由规则，所以可以将交换器理解成一个由 绑定 构成的路由表
- **Queue**：消息队列，用来保存消息到发送给消费者
  - 它是消息的容器，也是消息的终点
  - 一个消息可投入一个或多个队列
  - 消息一直在队列里面，等待消费者连接到这个队列将其取走
- **Consumer**：消费者，表示一个从消息队列中取得消息的客户端应用程序；
- **Message**：消息是没有名字的，它由 **消息头** 和 **消息体** 组成，消息体是不透明的，而消息头则由一系列的可选属性组成，这些属性包括：
  - routing-key（路由键）
  - priority（相对于其他消息的优先权）
  - delivery-mode（指出该消息可能需要持久性存储）等

三、RabbitMQ安装

> 官网：https://rabbitmq.com
>
> MQ下载地址：https://rabbitmq.com/download.html
>
> ErLang下载地址：https://packagecloud.io/rabbitmq/erlang/

（一）GCC 安装

> GCC 是 C 语言编辑环境

- 在线安装（方便）

> yum命令在线安装：`yum install gcc-c++`

```
[root@localhost redis-6.2.3]# yum install gcc-c++
已加载插件：fastestmirror, langpacks
Loading mirror speeds from cached hostfile
 * base: mirrors.aliyun.com
 * extras: mirrors.ustc.edu.cn
 * updates: mirrors.aliyun.com
base                                                                                                | 3.6 kB  00:00:00
extras                                                                                              | 2.9 kB  00:00:00
updates                                                                                             | 2.9 kB  00:00:00
(1/2): extras/7/x86_64/primary_db                                                                   | 236 kB  00:00:01
(2/2): updates/7/x86_64/primary_db                                                                  | 8.0 MB  00:00:16
正在解决依赖关系
--> 正在检查事务
---> 软件包 gcc-c++.x86_64.0.4.8.5-44.el7 将被 安装
--> 正在处理依赖关系 libstdc++-devel = 4.8.5-44.el7，它被软件包 gcc-c++-4.8.5-44.el7.x86_64 需要
--> 正在处理依赖关系 libstdc++ = 4.8.5-44.el7，它被软件包 gcc-c++-4.8.5-44.el7.x86_64 需要
--> 正在处理依赖关系 gcc = 4.8.5-44.el7，它被软件包 gcc-c++-4.8.5-44.el7.x86_64 需要
--> 正在检查事务
---> 软件包 gcc.x86_64.0.4.8.5-44.el7 将被 安装
--> 正在处理依赖关系 libgomp = 4.8.5-44.el7，它被软件包 gcc-4.8.5-44.el7.x86_64 需要
--> 正在处理依赖关系 cpp = 4.8.5-44.el7，它被软件包 gcc-4.8.5-44.el7.x86_64 需要
--> 正在处理依赖关系 libgcc >= 4.8.5-44.el7，它被软件包 gcc-4.8.5-44.el7.x86_64 需要
--> 正在处理依赖关系 glibc-devel >= 2.2.90-12，它被软件包 gcc-4.8.5-44.el7.x86_64 需要
---> 软件包 libstdc++.x86_64.0.4.8.5-36.el7 将被 升级
---> 软件包 libstdc++.x86_64.0.4.8.5-44.el7 将被 更新
---> 软件包 libstdc++-devel.x86_64.0.4.8.5-44.el7 将被 安装
--> 正在检查事务
---> 软件包 cpp.x86_64.0.4.8.5-44.el7 将被 安装
---> 软件包 glibc-devel.x86_64.0.2.17-324.el7_9 将被 安装
--> 正在处理依赖关系 glibc-headers = 2.17-324.el7_9，它被软件包 glibc-devel-2.17-324.el7_9.x86_64 需要
--> 正在处理依赖关系 glibc = 2.17-324.el7_9，它被软件包 glibc-devel-2.17-324.el7_9.x86_64 需要
--> 正在处理依赖关系 glibc-headers，它被软件包 glibc-devel-2.17-324.el7_9.x86_64 需要
---> 软件包 libgcc.x86_64.0.4.8.5-36.el7 将被 升级
---> 软件包 libgcc.x86_64.0.4.8.5-44.el7 将被 更新
```

```
  验证中      : libgcc-4.8.5-44.el7.x86_64                                                                 11/17
  验证中      : kernel-headers-3.10.0-1160.25.1.el7.x86_64                                                 12/17
  验证中      : glibc-2.17-260.el7.x86_64                                                                  13/17
  验证中      : libgomp-4.8.5-36.el7.x86_64                                                                14/17
  验证中      : libgcc-4.8.5-36.el7.x86_64                                                                 15/17
  验证中      : glibc-common-2.17-260.el7.x86_64                                                           16/17
  验证中      : libstdc++-4.8.5-36.el7.x86_64                                                              17/17

已安装:
  gcc-c++.x86_64 0:4.8.5-44.el7

作为依赖被安装:
  cpp.x86_64 0:4.8.5-44.el7                  gcc.x86_64 0:4.8.5-44.el7                        glibc-devel.x86_64 0:2.17-324.el7_9
  glibc-headers.x86_64 0:2.17-324.el7_9      kernel-headers.x86_64 0:3.10.0-1160.25.1.el7     libstdc++-devel.x86_64 0:4.8.5-44.el7

作为依赖被升级:
  glibc.x86_64 0:2.17-324.el7_9     glibc-common.x86_64 0:2.17-324.el7_9   libgcc.x86_64 0:4.8.5-44.el7   libgomp.x86_64 0:4.8.5-44.el7
  libstdc++.x86_64 0:4.8.5-44.el7

完毕!
[root@localhost redis-6.2.3]# gcc -v
使用内建 specs。
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/libexec/gcc/x86_64-redhat-linux/4.8.5/lto-wrapper
目标：x86_64-redhat-linux
配置为：../configure --prefix=/usr --mandir=/usr/share/man --infodir=/usr/share/info --with-bugurl=http://bugzilla.redhat.com/bugzilla --enable-bootstrap --enable-shared --enable-threads=posix --enable-checking=release --with-system-zlib --enable-__cxa_atexit --disable-libunwind-exceptions --enable-gnu-unique-object --enable-linker-build-id --with-linker-hash-style=gnu --enable-languages=c,c++,objc,obj-c++,java,fortran,ada,go,lto --enable-plugin --enable-initfini-array --disable-libgcj --with-isl=/builddir/build/BUILD/gcc-4.8.5-20150702/obj-x86_64-redhat-linux/isl-install --with-cloog=/builddir/build/BUILD/gcc-4.8.5-20150702/obj-x86_64-redhat-linux/cloog-install --enable-gnu-indirect-function --with-tune=generic --with-arch_32=x86-64 --build=x86_64-redhat-linux
线程模型：posix
gcc 版本 4.8.5 20150623 (Red Hat 4.8.5-44) (GCC)
[root@localhost redis-6.2.3]#
```

- 离线安装

> 将Linux的系统文件加载到光驱，切换到目录：`cd /run/media/root/CentOS 7 x86_64/Packages/`，将一下命令挨个执行即可，注意不同的系统镜像版本不同。

```
rpm -ivh cpp-4.8.5-11.el7.x86_64.rpm
rpm -ivh kernel-headers-3.10.0-514.el7.x86_64.rpm
rpm -ivh glibc-headers-2.17-157.el7.x86_64.rpm
rpm -ivh glibc-devel-2.17-157.el7.x86_64.rpm
rpm -ivh libgomp-4.8.5-11.el7.x86_64.rpm
rpm -ivh gcc-4.8.5-11.el7.x86_64.rpm
```

2、安装`rpm -ivh cpp-4.8.5-11.el7.x86_64.rpm`报错解决放案：

```
输入命令：rpm -ivh libmpc-1.0.1-3.el7.x86_64.rpm
```

（二）RabbitMQ 安装

1、安装 openssl

> openssl 是 erlang 软件所需要的依赖

```
openssl-1.1.1l.tar.gz
```

```
# 解压
tar -zxvf openssl-1.1.1l.tar.gz

# 安装，依次输入以下三个命令即可
./config
make && make install
./config shared

make clean
make && make install
```

> 或者使用 rpm 安装也可以

```
rpm -ivh openssl-libs-1.0.2k-19.el7.x86_64.rpm --force
```

2、安装 Erlang

> RabbitMQ 依赖语言

```
rpm -ivh erlang-23.3.4.5-1.el7.x86_64.rpm
```

3、安装 socat

> RabbitMQ 依赖包

```
yum install socat logrotate -y
```

4、安装 RabbitMQ

```
rpm -ivh rabbitmq-server-3.9.4-1.el7.noarch.rpm
```

5、安装 RabbitMQ Web插件

```
# 开启web插件
rabbitmq-plugins enable rabbitmq_management
```

```
[root@localhost mq-soft]# rabbitmq-plugins enable rabbitmq_management
Enabling plugins on node rabbit@localhost:
rabbitmq_management
The following plugins have been configured:
  rabbitmq_management
  rabbitmq_management_agent
  rabbitmq_web_dispatch
Applying plugin configuration to rabbit@localhost...
The following plugins have been enabled:
  rabbitmq_management
  rabbitmq_management_agent
  rabbitmq_web_dispatch

set 3 plugins.
Offline change; changes will take effect at broker restart.
[root@localhost mq-soft]#
```

6、启动关闭 RabbitMQ

```
# 在 /sbin 目录下可以看到已经安装的 RabbitMQ 的命令
```

```
-rwxr-xr-x. 1 root root        2759 8月   18 20:53 rabbitmq-diagnostics
-rwxr-xr-x. 1 root root        2759 8月   18 20:53 rabbitmq-plugins
-rwxr-xr-x. 1 root root        2759 8月   18 20:53 rabbitmq-queues
-rwxr-xr-x. 1 root root        2759 8月   18 20:53 rabbitmq-server
-rwxr-xr-x. 1 root root        2759 8月   18 20:53 rabbitmq-streams
-rwxr-xr-x. 1 root root        2759 8月   18 20:53 rabbitmq-upgrade
```

```
# 查看MQ启动状态
service rabbitmq-server status
# 启动MQ
service rabbitmq-server start
# 关闭MQ
service rabbitmq-server stop
```

如图：active（running）标识服务正在运行

```
[root@localhost rabbitmq]# service rabbitmq-server start
Redirecting to /bin/systemctl start rabbitmq-server.service
[root@localhost rabbitmq]# service rabbitmq-server status
Redirecting to /bin/systemctl status rabbitmq-server.service
● rabbitmq-server.service - RabbitMQ broker
   Loaded: loaded (/usr/lib/systemd/system/rabbitmq-server.service; disabled; vendor preset: disabled)
   Active: active (running) since 一 2021-08-30 22:51:25 CST; 9s ago
 Main PID: 32937 (beam.smp)
    Tasks: 21
   CGroup: /system.slice/rabbitmq-server.service
           ├─32937 /usr/lib64/erlang/erts-11.2.2.4/bin/beam.smp -W w -MBas ageffcbf -MHas ageffcbf -MBlmbcs 512 -MHlmbcs 512 -MMmcs 30 -P 1048576 -t 500000...
           ├─32952 erl_child_setup 32768
           ├─32992 inet_gethost 4
           └─32993 inet_gethost 4

8月 30 22:51:24 localhost.localdomain rabbitmq-server[32937]: Doc guides:  https://rabbitmq.com/documentation.html
8月 30 22:51:24 localhost.localdomain rabbitmq-server[32937]: Support:     https://rabbitmq.com/contact.html
8月 30 22:51:24 localhost.localdomain rabbitmq-server[32937]: Tutorials:   https://rabbitmq.com/getstarted.html
8月 30 22:51:24 localhost.localdomain rabbitmq-server[32937]: Monitoring:  https://rabbitmq.com/monitoring.html
8月 30 22:51:24 localhost.localdomain rabbitmq-server[32937]: Logs: /var/log/rabbitmq/rabbit@localhost.log
8月 30 22:51:24 localhost.localdomain rabbitmq-server[32937]: /var/log/rabbitmq/rabbit@localhost_upgrade.log
8月 30 22:51:24 localhost.localdomain rabbitmq-server[32937]: <stdout>
8月 30 22:51:24 localhost.localdomain rabbitmq-server[32937]: Config file(s): (none)
8月 30 22:51:25 localhost.localdomain systemd[1]: Started RabbitMQ broker.
8月 30 22:51:25 localhost.localdomain rabbitmq-server[32937]: Starting broker... completed with 3 plugins.
```

7、通过浏览器访问

> 浏览器输入：http://192.168.88.88:15672
>
> 前提是Linux 防火墙允许15672端口或者关闭防火墙，建议放行15672端口即可。

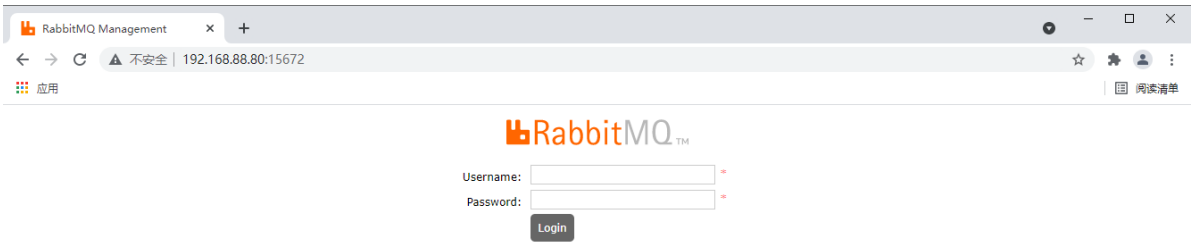
- **将15672/5672端口放行**

注意：配置端口之前先通过 `service rabbitmq-server stop` 关闭服务

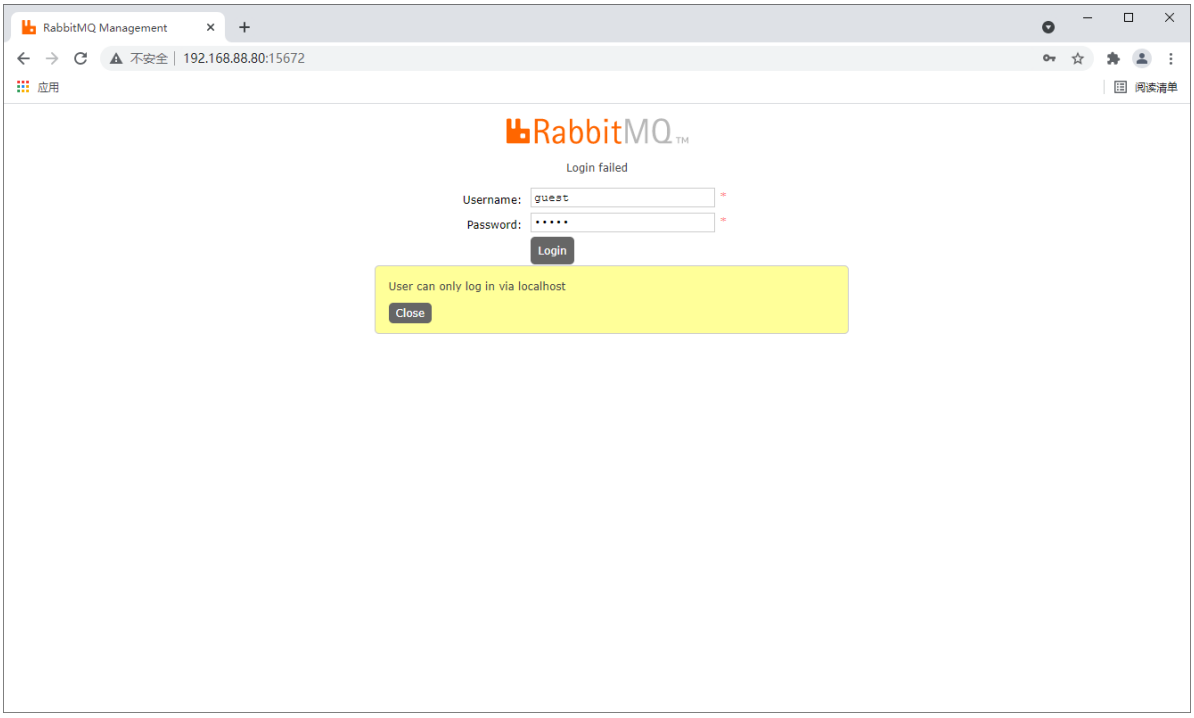
```
# 添加端口到防火墙
firewall-cmd --zone=public --add-port=15672/tcp --permanent
firewall-cmd --zone=public --add-port=5672/tcp --permanent
# 重新加载防火墙
firewall-cmd --reload
# 验证端口是否打开
firewall-cmd --zone=public --list-port
# ----------------------------------------------------------------
# 关闭防火墙
systemctl stop firewalld
```

```
[root@localhost mq-soft]# firewall-cmd --zone=public --add-port=15672/tcp --permanent
success
[root@localhost mq-soft]# firewall-cmd --reload
success
[root@localhost mq-soft]# firewall-cmd --zone=public --list-port
15672/tcp
[root@localhost mq-soft]#
```

- **访问页面**



> 使用默认 `guest` 账号登录，默认情况下`guest`仅支持本地访问，如何解决？看下一步。



- **配置 RabbitMQ 支持远程登录**

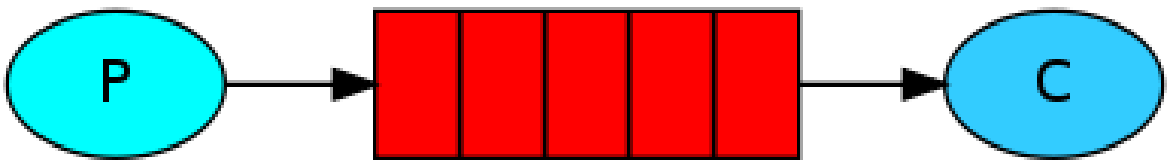
创建个人账号赋予权限实现远程登录

```
# 创建账号
rabbitmqctl add_user 账号 密码
# eg:
rabbitmqctl add_user admin admin

# 设置账号角色
rabbitmqctl set_user_tags 账号 角色
# eg:
rabbitmqctl set_user_tags admin administrator

# 设置角色权限
rabbitmqctl set_permissions [-p <vhostpath>] <user> <conf> <write> <read>
# eg:
rabbitmqctl set_permissions -p "/" admin ".*" ".*" ".*"

# 查看账号列表
rabbitmqctl list_users
```

```
[root@localhost rabbitmq]# rabbitmqctl list_users
Listing users ...
user    tags
admin   [administrator]
guest   [administrator]
[root@localhost rabbitmq]#
```

四、简单队列模式

> 简单队列模式一般指一个生产者和一个消费者，生产者生产消息，推送到队列中，被消费者消费获取。



1、导入依赖

```xml
<dependency>
    <groupId>com.rabbitmq</groupId>
    <artifactId>amqp-client</artifactId>
    <version>5.15.0</version>
</dependency>
```

2、创建一个生产者

```java
/**
 * 生产者：生产消息
 */
public class HelloProducer {

    private static String QUEUE_NAME = "Hello";

    public static void main(String[] args) throws Exception {
        // 1、通过连接工厂
        ConnectionFactory factory = new ConnectionFactory();

        // 设置 MQ 的地址
        factory.setHost("192.168.88.88");
        // 设置 MQ 端口号
        factory.setPort(5672);
        // 设置账号密码
        factory.setUsername("admin");
        factory.setPassword("123456");

        // 2、构建连接
        Connection connection = factory.newConnection();
        // 3、构建信道
        Channel channel = connection.createChannel();

        // 4、生成一个队列
        /*
            String queue：队列名称
            boolean durable：队列中的消息是否需要持久化，默认队列中的消息存储在内存中，即 false
            boolean exclusive：当前队列是否只供一个消费者使用。true 表示可以多个消费者使用
            boolean autoDelete：最后一个消费者断开连接之后，该队列是否自动删除。true 表示自动删除
            Map<String, Object> arguments：其他参数
         */
        channel.queueDeclare(QUEUE_NAME, false, false, false, null);

        // 5、发布消息
        String msg = "Hello World";

        /*
            String exchange：交换机，这里使用简单模式，没有交换机
            String routingKey：路由键，默认路由没有 routingKey，这里使用队列名称代替。
            BasicProperties props：参数，这里使用简单模式，没有参数
```

```
            byte[] body：消息体
         */
        channel.basicPublish("", QUEUE_NAME, null, msg.getBytes("UTF-8"));
        System.out.println("消息生产成功！");

        // 6、关闭连接
        channel.close();
        connection.close();
    }
}
```

3、创建一个消费者

写法一：

```
/**
 * 消费者：接收消息
 */
public class HelloConsumer {

    private static String QUEUE_NAME = "Hello";

    public static void main(String[] args) throws Exception {
        // 1、通过连接工厂
        ConnectionFactory factory = new ConnectionFactory();

        // 设置 MQ 的地址
        factory.setHost("192.168.88.88");
        // 设置 MQ 端口号
        factory.setPort(5672);
        // 设置账号密码
        factory.setUsername("admin");
        factory.setPassword("123456");

        // 2、构建连接
        Connection connection = factory.newConnection();
        // 3、构建信道
        Channel channel = connection.createChannel();

        // 4、接收消息
        DefaultConsumer consumer = new DefaultConsumer(channel){
            /*
                String consumerTag
                Envelope envelope
                AMQP.BasicProperties properties
                byte[] body：消息体；消息类型多样，可能是文字、视频、图片，故采用字节形式。
            */
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, byte[] body) throws IOException {
                // 由于生产者存放消息是文字，所以需要将字节数组转化为字符串。使用 String 的构造方法即可。
                String msg = new String(body, "UTF-8");
                System.out.println("【消费者接收消息】" + msg);
```

```
            }
        };
        System.out.println("消息接收中...");
        /*
            String queue：队列名称
            boolean autoAck：接收消息之后自动应答。true 自动应答
            Consumer callback：消费者回调方法
         */
        channel.basicConsume(QUEUE_NAME, true, consumer);

        // 5、关闭连接
        channel.close();
        connection.close();
    }
}
```

写法二：

```
/**
 * 消费者：接收消息
 */
public class HelloConsumer {

    private static String QUEUE_NAME = "Hello";

    public static void main(String[] args) throws Exception {
        // 1、通过连接工厂
        ConnectionFactory factory = new ConnectionFactory();

        // 设置 MQ 的地址
        factory.setHost("192.168.88.88");
        // 设置 MQ 端口号
        factory.setPort(5672);
        // 设置账号密码
        factory.setUsername("admin");
        factory.setPassword("123456");

        // 2、构建连接
        Connection connection = factory.newConnection();
        // 3、构建信道
        Channel channel = connection.createChannel();

        // 4、接收消息

        // 接收消息的回调对象
        DeliverCallback deliverCallback = (consumerTag, message) -> {
            byte[] body = message.getBody();
            String msg = new String(body);
            System.out.println("接收到的消息是：" + msg);
        };

        // 取消消息的回调对象
        CancelCallback cancelCallback = consumerTag -> {
        };

```

```
        /*
            String queue：队列名称
            boolean autoAck：接收消息之后自动应答。true 自动应答
            DeliverCallback deliverCallback：接收消息的回调函数
            CancelCallback cancelCallback：取消接收消息的回调函数
         */
        channel.basicConsume(QUEUE_NAME, true, deliverCallback,
cancelCallback);

        // 5、关闭连接
        channel.close();
        connection.close();
    }
}
```

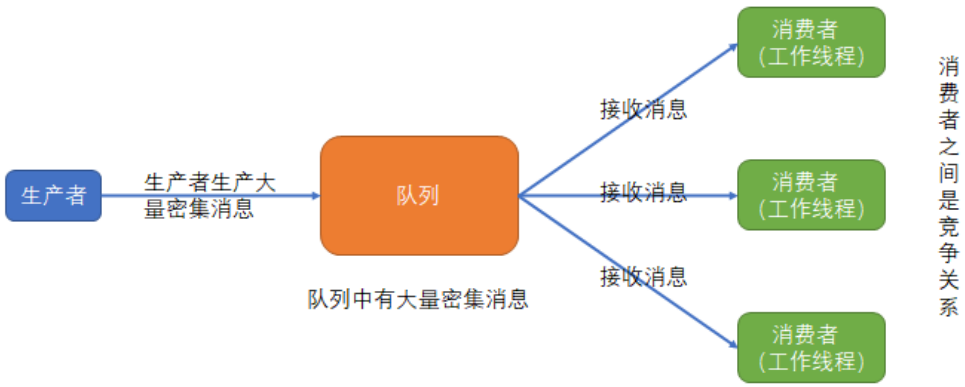
4、封装工具类

```
package com.soft.util;

import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;

/**
 * RabbitMQ 工具类
 * @author riu
 */
public class RabbitMqUtil {

    private String HOST = "192.168.88.88";
    private static Integer PORT = 5672;
    private static String USERNAME = "admin";
    private static String PASSWORD = "123456";

    Connection connection = null;
    Channel channel = null;

    /**
     * 创建信道
     */
    public Channel getChannel(){
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost(HOST);
        factory.setPort(PORT);
        factory.setUsername(USERNAME);
        factory.setPassword(PASSWORD);

        try{
            connection = factory.newConnection();
            channel = connection.createChannel();

            return channel;

        } catch (Exception e){
            System.out.println("信道创建失败!");
```

```
        }
        return null;
    }

    /**
     * 释放资源
     */
    public void close(){
        try {
            /* 注意：先开的后关 */
            if(channel != null){
                channel.close();
            }

            if(connection != null){
                connection.close();
            }
        } catch (Exception e){
            System.out.println("资源释放异常！");
        }
    }
}
```

五、工作队列模式

> 工作队列模式一般指一个生产者多个消费者，生产者生产大量密集消息，放入队列。消费者有序或者无序从队列中获取消息。每个消息只能被接收一次，不允许被接收多次。所以多个消费者要进行轮巡，消费者之间是竞争关系，这也是工作队列模式默认的工作方式。



（一）轮询接收

1、创建生产者

```
public class WorkProducer {

    private static String QUEUE_NAME = "Work";
```

```
4
5       public static void main(String[] args) throws Exception {
6           // 1、调用工具类生成信道
7           RabbitMqUtil rabbitMqUtil = new RabbitMqUtil();
8           Channel channel = rabbitMqUtil.getChannel();
9
10          /*
11              2、生成队列
12              String queue：队列名称
13              boolean durable：队列中的消息是否需要持久化，默认队列中的消息存储在内存
    中，即 false
14              boolean exclusive：当前队列是否只供一个消费者使用。true 表示可以多个消费
    者使用
15              boolean autoDelete：最后一个消费者断开连接之后，该队列是否自动删除。true
    表示自动删除
16              Map<String, Object> arguments：其他参数
17           */
18          channel.queueDeclare(QUEUE_NAME, false, false, false, null);
19
20          /*
21              3、发布消息
22              String exchange：交换机，这里使用简单模式，没有交换机
23              String routingKey：队列名称
24              BasicProperties props：参数，这里使用简单模式，没有参数
25              byte[] body：消息体
26           */
27          for (int i = 0; i < 10; i++) {
28              String msg = "工作模式！" + i;
29              channel.basicPublish("", QUEUE_NAME, null, msg.getBytes());
30          }
31          System.out.println("消息生产成功！");
32
33          // 4、调用工具类释放资源
34          rabbitMqUtil.close();
35      }
36  }
37
```

2、创建消费者

- 消费者1

```
1   public class WorkConsumer1 {
2
3       private static String QUEUE_NAME = "Work";
4
5       public static void main(String[] args) throws Exception {
6           // 1、调用工具类生成信道
7           RabbitMqUtil rabbitMqUtil = new RabbitMqUtil();
8           Channel channel = rabbitMqUtil.getChannel();
9           /*
10              2、接收消息
11           */
12          DefaultConsumer consumer = new DefaultConsumer(channel){
13              @Override
```

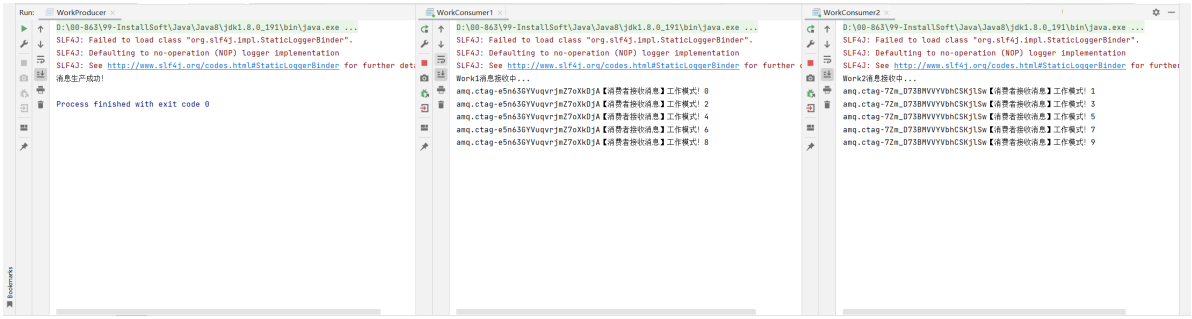
```
            public void handleDelivery(String consumerTag, Envelope
envelope, AMQP.BasicProperties properties, byte[] body) throws IOException {
                String msg = new String(body);
                System.out.println(consumerTag + "【消费者接收消息】" + msg);
                try {
                    Thread.sleep(2000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        };
        System.out.println("Work1消息接收中...");
        channel.basicConsume(QUEUE_NAME, true, consumer);

        // 这里为了验证多消费者接收信息，暂时不释放资源
        // rabbitMqUtil.close();
    }
}

```

- 消费者2

```
public class WorkConsumer2 {

    private static String QUEUE_NAME = "Work";

    public static void main(String[] args) throws Exception {
        RabbitMqUtil rabbitMqUtil = new RabbitMqUtil();
        Channel channel = rabbitMqUtil.getChannel();
        /*
            接收消息
         */
        DefaultConsumer consumer = new DefaultConsumer(channel){
            @Override
            public void handleDelivery(String consumerTag, Envelope
envelope, AMQP.BasicProperties properties, byte[] body) throws IOException {
                String msg = new String(body);
                System.out.println(consumerTag + "【消费者接收消息】" + msg);
            }
        };
        System.out.println("Work2消息接收中...");
        channel.basicConsume(QUEUE_NAME, true, consumer);

        // 这里为了验证多消费者接收信息，暂时不释放资源
        // rabbitMqUtil.close();
    }
}

```

3、结果

> 生产者生成消息，消费者1和消费者2挨个获取消息，即便是消费者1存在摸鱼现象。



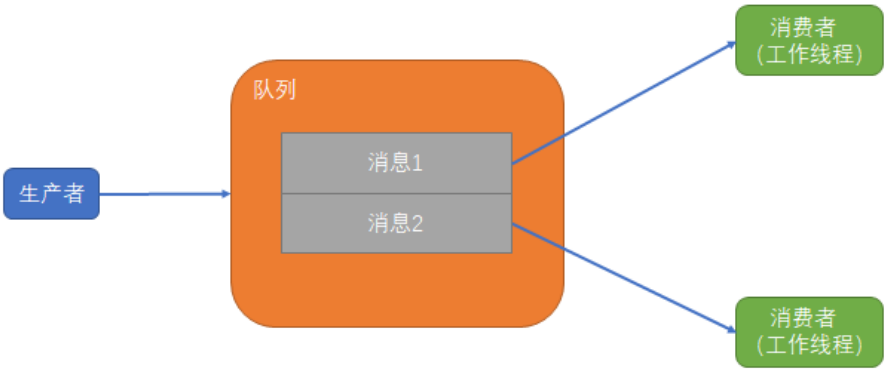
（二）消息应答

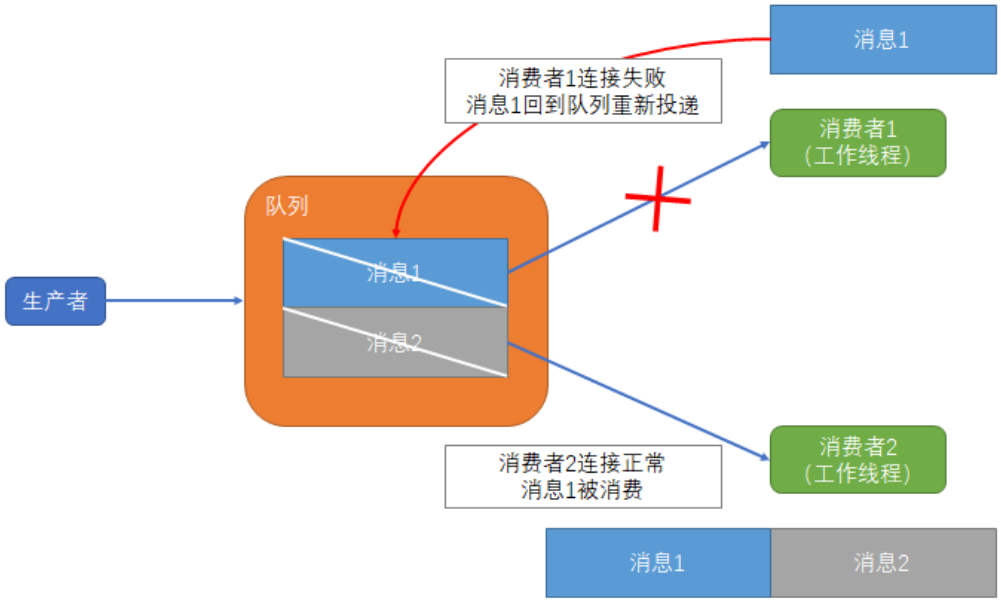
> 消费者完成一个任务可能需要一段时间，如果其中一个消费者处理一个复杂的任务并仅只完成了部分突然它挂掉了，会产生什么结果?
>
> RabbitMQ 一旦向消费者传递了一条消息，当消费者接收到消息便立即将该消息标记为删除。在这种情况下，如果个消费者挂掉了，将丢失正在处理的消息，同时后续发送给该消费这的消息，也无法接收到。
>
> **为了保证消息在发送过程中不丢失**，RabbitMQ 引入**消息应答机制**，即消费者在接收到消息并且处理该消息之后，告诉 RabbitMQ 它已经处理了，RabbitMQ 可以把消息删除了。

1、自动应答

> 自动应答默认是开启的，MQ 不在乎消费者对消息处理是否成功，只要接收到消息就会告诉队列删除当前消息。如果后续处理失败，为了防止数据丢失，MQ 实现自动重新投递策略，重新将消息发送，并且让有能力处理的消费者处理（使用手动应答）。

重新投递流程:





2、手动应答

> 消费者从队列中获取消息后，服务器会将该消息标记为不可用状态，等待消费者的反馈，如果消费者一直没有反馈，那么该消息将一直处于不可用状态。

```
// 用于肯定消息：MQ 已经知道该消息被消费者成功接收，并且处理完成，可以将消息丢弃
channel.basicAck();

// 用于否定确认：
channel.basicNack();

// 用于否定确认：
channel.basicReject();
```

（1）创建生产者

```
/**
 * 生产者：生产消息
 */
public class UnAutoAckProducer {

    public static final String QUEUE_NAME = "UnAutoAck";

    public static void main(String[] args) throws Exception {
        // 1、调用工具类生成信道
        RabbitMqUtil rabbitMqUtil = new RabbitMqUtil();
        Channel channel = rabbitMqUtil.getChannel();
        /*
            2、生成一个队列
            String queue：队列名称
            boolean durable：队列中的消息是否需要持久化，默认队列中的消息存储在内存中，即 false
```

```
            boolean exclusive：当前队列是否只供一个消费者使用。true 表示可以多个消费者使用
            boolean autoDelete：最后一个消费者断开连接之后，该队列是否自动删除。true 表示自动删除
            Map<String, Object> arguments：其他参数
         */
        channel.queueDeclare(QUEUE_NAME, false, false, false, null);

        // 3、发布消息
        // 通过控制台循环输入消息
        Scanner scanner = new Scanner(System.in);
        while (true) {
            System.out.print("输入消息：");
            String msg = scanner.next();

            /*
                String exchange：交换机，这里使用简单模式，没有交换机
                String routingKey：队列名称
                BasicProperties props：参数，这里使用简单模式，没有参数
                byte[] body：消息体
             */
            channel.basicPublish("", QUEUE_NAME, null, msg.getBytes());
            System.out.println("消息" + msg + "发送成功！");
        }
    }
}
```

（2）创建消费者

消费者C1：

```
/**
 * 消费者：接收消息
 */
public class UnAutoAckConsumer1 {
    public static void main(String[] args) throws Exception {
        // 1、调用工具类生成信道
        RabbitMqUtil rabbitMqUtil = new RabbitMqUtil();
        Channel channel = rabbitMqUtil.getChannel();
        /*
            2、接收消息
         */
        DefaultConsumer consumer = new DefaultConsumer(channel){
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, byte[] body) throws IOException {
                String msg = new String(body);
                System.out.println(consumerTag + "【消费者接收消息】" + msg);

                try {
                    Thread.sleep(2000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

```

```
                // 手动应答
                channel.basicAck(envelope.getDeliveryTag(), false);
            }
        };
        System.out.println("Work1消息接收中...");
        // 配置关闭自动应答
        channel.basicConsume(QUEUE_NAME, false, consumer);

        // 这里为了验证多消费者接收信息，暂时不释放资源
        // rabbitMqUtil.close();
    }
}

```

消费者C2：

```
/**
 * 消费者：接收消息
 */
public class UnAutoAckConsumer1 {
    public static void main(String[] args) throws Exception {
        // 1、调用工具类生成信道
        RabbitMqUtil rabbitMqUtil = new RabbitMqUtil();
        Channel channel = rabbitMqUtil.getChannel();
        /*
            2、接收消息
         */
        DefaultConsumer consumer = new DefaultConsumer(channel){
            @Override
            public void handleDelivery(String consumerTag, Envelope
envelope, AMQP.BasicProperties properties, byte[] body) throws IOException {
                String msg = new String(body);
                System.out.println(consumerTag + "【消费者接收消息】" + msg);

                // 手动应答
                channel.basicAck(envelope.getDeliveryTag(), false);
            }
        };
        System.out.println("Work2消息接收中...");
        // 配置关闭自动应答
        channel.basicConsume(QUEUE_NAME, false, consumer);

        // 这里为了验证多消费者接收信息，暂时不释放资源
        // rabbitMqUtil.close();
    }
}

```

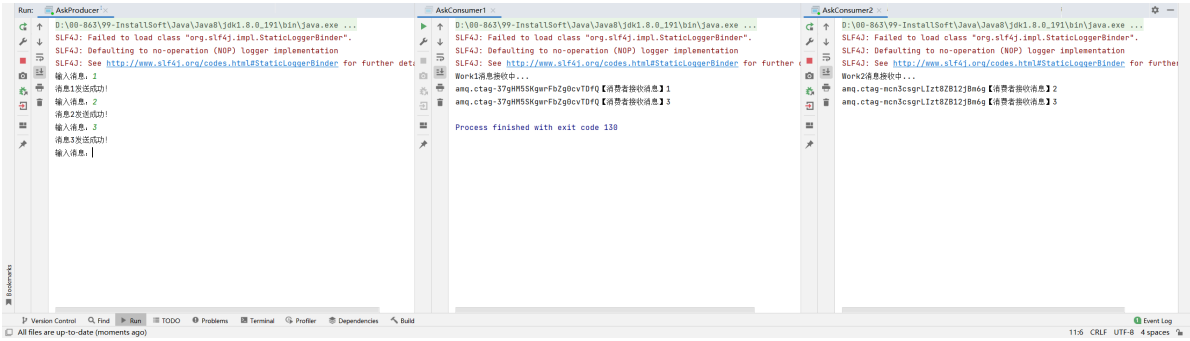
（3）结果

- 情景1：

```
生产者：生产消息

消费者1（处理时间长）：开启线程，正常接收
消费者2（处理时间短）：开启线程，正常接收

最终结果：消费者1、2均接收正常
```

- 情景2：

```
生产者：生产消息

消费者1（处理时间长）：开启线程，断开连接
消费者2（处理时间短）：开启线程，正常接收

最终结果：
    消费者2正常接收消息
    消费者1断开连接，本该消费者1消费的消息回到队列重新分配，由于消费者2处理时间较短，此时处于闲置状态，所以消息由消费者2消费并应答。
```



（三）能者多劳

> MQ 默认情况下要求消费者挨个获取消息，即便是存在某个消费者存在摸鱼的情况，那么这种情况是不明智的。我们希望看到的是能者多劳。实现能者多劳只需配置每个消费者同时只能接收一个消息`channel.basicQos(1)`，其本质是设置`prefetchCount`预读取数量。另外，还要求手动应答。

1、创建生产者

```java
public class WorkUnAvgProducer {

    private static String QUEUE_NAME = "Work";

    public static void main(String[] args) throws Exception {
        // 1、调用工具类生成信道
        RabbitMqUtil rabbitMqUtil = new RabbitMqUtil();
        Channel channel = rabbitMqUtil.getChannel();

        /*
            2、生成队列
            String queue：队列名称
            boolean durable：队列中的消息是否需要持久化，默认队列中的消息存储在内存中，即 false
            boolean exclusive：当前队列是否只供一个消费者使用。true 表示可以多个消费者使用
```

```
            boolean autoDelete：最后一个消费者断开连接之后，该队列是否自动删除。true表示自动删除
            Map<String, Object> arguments：其他参数
         */
        channel.queueDeclare(QUEUE_NAME, false, false, false, null);

        /*
            3、发布消息
            String exchange：交换机，这里使用简单模式，没有交换机
            String routingKey：队列名称
            BasicProperties props：参数，这里使用简单模式，没有参数
            byte[] body：消息体
         */
        for (int i = 0; i < 10; i++) {
            String msg = "工作模式！" + i;
            channel.basicPublish("", QUEUE_NAME, null, msg.getBytes());
        }
        System.out.println("消息生产成功！");

        // 4、释放资源
        rabbitMqUtil.close();
    }
}
```

2、创建消费者

- 消费者1

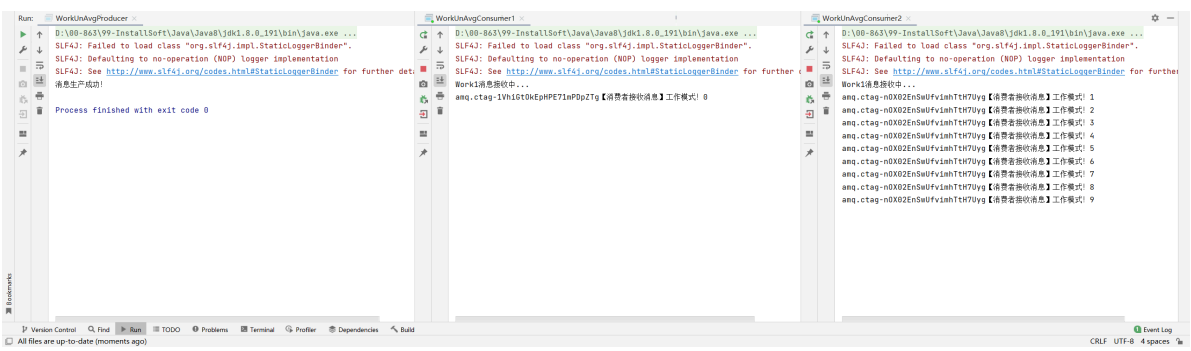
```
public class WorkUnAvgConsumer1 {

    private static String QUEUE_NAME = "Work";

    public static void main(String[] args) throws Exception {
        // 1、调用工具类生成信道
        RabbitMqUtil rabbitMqUtil = new RabbitMqUtil();
        Channel channel = rabbitMqUtil.getChannel();
        // 配置预读取数量
        channel.basicQos(1);
        /*
            2、接收消息
         */
        DefaultConsumer consumer = new DefaultConsumer(channel){
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, byte[] body) throws IOException {
                String msg = new String(body);
                System.out.println(consumerTag + "【消费者接收消息】" + msg);
                try {
                    Thread.sleep(2000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                // 手动应答
                channel.basicAck(envelope.getDeliveryTag(), false);
            }
```

```
        };
        System.out.println("Work1消息接收中...");
        // 取消自动应答
        channel.basicConsume(QUEUE_NAME, false, consumer);

        // rabbitMqUtil.close();
    }
}

```

- 消费者2

```
public class WorkUnAvgConsumer2 {

    private static String QUEUE_NAME = "Work";

    public static void main(String[] args) throws Exception {
        // 1、调用工具类生成信道
        RabbitMqUtil rabbitMqUtil = new RabbitMqUtil();
        Channel channel = rabbitMqUtil.getChannel();
        // 配置预读取数量
        channel.basicQos(1);
        /*
            2、接收消息
         */
        DefaultConsumer consumer = new DefaultConsumer(channel){
            @Override
            public void handleDelivery(String consumerTag, Envelope
envelope, AMQP.BasicProperties properties, byte[] body) throws IOException {
                String msg = new String(body);
                System.out.println(consumerTag + "【消费者接收消息】" + msg);
                // 手动应答
                channel.basicAck(envelope.getDeliveryTag(), false);
            }
        };
        System.out.println("Work1消息接收中...");
        // 取消自动应答
        channel.basicConsume(QUEUE_NAME, false, consumer);

        // rabbitMqUtil.close();
    }
}

```

3、结果



六、发布确认

（一）概念

> 消息在传递的过程中存在丢失的肯能，所以 RabbitMQ 提出了**持久化**操作，将生产者生产的消息**持久化到硬盘**，以保证消息不丢失进行正常传输。
>
> 要求**队列持久化、消息持久化。**
>
> 进行了持久化之后也并不一定确保消息不会丢失，很有可能在持久化的过程中消息丢失。所以又提出发布确认模式。
>
> 发布确认要求**队列持久化、消息持久化**，同时持久化到硬盘之后给生产者反馈信息，以保证消息真正持久化到硬盘。

（二）创建生产者

1、队列持久化、消息持久化

- 队列持久化

```
/*
    String queue：队列名称
    boolean durable：队列中的消息是否需要持久化，默认队列中的消息存储在内存中，即 false
    boolean exclusive：当前队列是否只供一个消费者使用。true 表示可以多个消费者使用
    boolean autoDelete：最后一个消费者断开连接之后，该队列是否自动删除。true 表示自动删除
    Map<String, Object> arguments：其他参数
*/
// 修改 durable 为 true
channel.queueDeclare(QUEUE_NAME, true, false, false, null);
```

PS：信道在第一次创建时没有持久化，第二次创建持久化时会报错，需要删掉之后重新创建。

```
Caused by: com.rabbitmq.client.ShutdownSignalException: channel error; protocol method: #method<channel.close>(reply-code=406, reply-text=PRECONDITION_FAILED - inequivalent arg 'durable' for queue '信道名字' in vhost '/': received 'true' but current is 'false', class-id=50, method-id=10)
```

Features 显示为大写的D表示持久化信道。

Overview					Messages					Message bytes					Message rates		
Name	Type	Features	Consumer capacity	State	Ready	Unacked	In Memory	Persistent	Total	Ready	Unacked	In Memory	Persistent	Total	incoming	deliver / get	ack
Ask	classic		0%	idle	0	0	0	0	0	0 B	0 B	0 B	0 B	0 B	0.00/s	0.00/s	0.00/s
Work	classic	D	0%	idle	10	0	10	0	10	160 B	0 B	160 B	0 B	160 B	0.00/s		

- 消息持久化

```
/*
    String exchange：交换机，这里使用简单模式，没有交换机
    String routingKey：队列名称
    BasicProperties props：参数，这里 MessageProperties.PERSISTENT_TEXT_PLAIN 让消息持久化
    byte[] body：消息体
*/
// 添加参数：MessageProperties.PERSISTENT_TEXT_PLAIN
channel.basicPublish("", QUEUE_NAME, MessageProperties.PERSISTENT_TEXT_PLAIN, msg.getBytes());
```

2、开启确认发布

```
// 构建信道
Channel channel = connection.createChannel();
// 开启发布确认
channel.confirmSelect();
```

3、单个发布确认

> 单消息发布确认是最简单的确认方式，是一种同步确认发布的方式，也就是发布消息之后只有被确认发布，后续的消息才能继续发布。最大的缺点就是：发布速度慢，效率低。

```
/**
 * 生产者：生产消息
 */
public class ConfirmOnceProducer {

    public static final String QUEUE_NAME = "Confirm";

    public static void main(String[] args) throws Exception {
        // 1、工具类创建信道
        RabbitMqUtil rabbitMqUtil = new RabbitMqUtil();
        Channel channel = rabbitMqUtil.getChannel();
        // 2、开启发布确认
        channel.confirmSelect();
        // 3、生成一个队列，并持久化信道
        channel.queueDeclare(QUEUE_NAME, true, false, false, null);

        // 4、发布消息
        long start = System.currentTimeMillis();
        for (int i = 0; i < 1000; i++) {
            String msg = i + "";

            // 消息持久化：MessageProperties.PERSISTENT_TEXT_PLAIN
            channel.basicPublish("", QUEUE_NAME,
MessageProperties.PERSISTENT_TEXT_PLAIN, msg.getBytes());
            // 等待确认
            boolean confirm = channel.waitForConfirms();

            if(confirm){
                System.out.println("消息" + msg + "发送成功！");
            }
        }
        long end = System.currentTimeMillis();
        System.out.println("单个发布确认消耗时间：" + (end - start));
    }
}
```

4、批量发布确认

> 批量发布确认比单个发布确认的效率要高，但也存在一些问题。一批数据发布之后，中间某个消息传输失败时并不能定位到具体消息。为了保证消息可以全部发送，必须将整批次的数据保存起来，重新发布。

```

```

```java
/**
 * 生产者：生产消息
 */
public class ConfirmBatchProducer {

    public static final String QUEUE_NAME = "Confirm";

    public static void main(String[] args) throws Exception {
        // 1、工具类创建信道
        RabbitMqUtil rabbitMqUtil = new RabbitMqUtil();
        Channel channel = rabbitMqUtil.getChannel();
        // 2、开启发布确认
        channel.confirmSelect();
        // 3、生成一个队列，并持久化信道
        channel.queueDeclare(QUEUE_NAME, true, false, false, null);

        // 4、发布消息
        long start = System.currentTimeMillis();
        // 定义确认节点
        int count = 1000;
        for (int i = 0; i < 5000; i++) {
            String msg = i + "";

            // 消息持久化：MessageProperties.PERSISTENT_TEXT_PLAIN
            channel.basicPublish("", QUEUE_NAME,
MessageProperties.PERSISTENT_TEXT_PLAIN, msg.getBytes());


            if(i % count == 0){
                // 等待确认
                boolean confirm = channel.waitForConfirms();
                if(confirm){
                    System.out.println("消息" + msg + "发送成功！");
                }
            }
        }
        long end = System.currentTimeMillis();
        System.out.println("批量发布确认消耗时间：" + (end - start));
    }
}
```

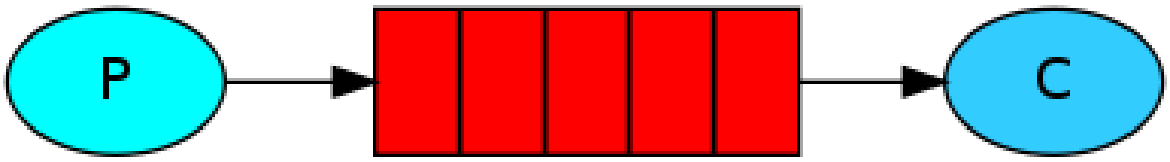
七、交换机

> 简单模式和工作队列模式实现生产者生产消息，每个消息只能由一个消费者接收（不配置自动应答和手动应答的情况另说），也就是说，消费者之间属于竞争关系。
>
> 如果要实现生产者生产消息，消息可以供多个消费者同时接收，这样的场景就要用到交换机。
>
> 简单来说，之前消息放在一个队列中，仅供一个消费者消费。如果消息需要被多个消费者消费，可以将消息放在多个队列中，每个队列对应指定的消费者，即可实现消息共享。那么，消息如何存放到指定的队列中或者丢弃，就需要交换机。

（一）交换机概念

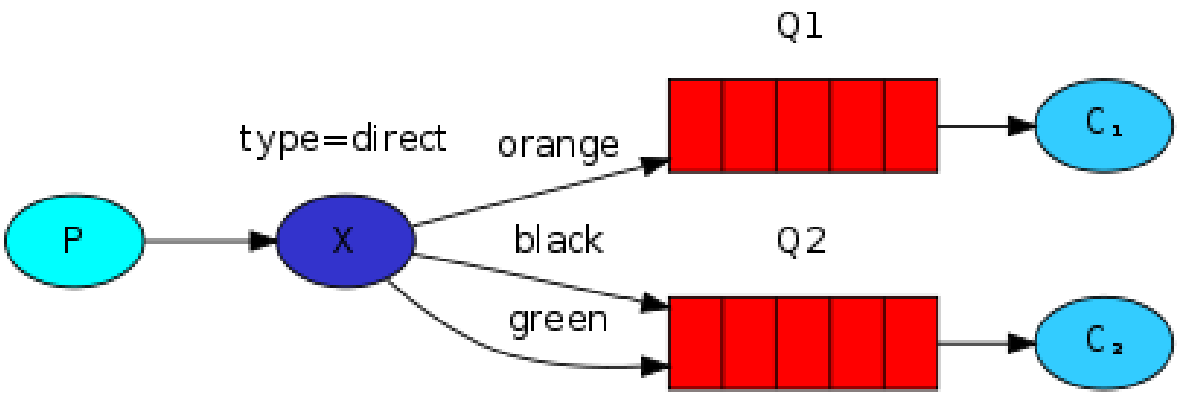
> 交换机（Exchange）在 RabbitMQ 中起到一个信息中转的作用，生产者生产的消息都是通过交换机发送给队列，简单模式和工作队列模式中并没有特别强调交换机，是因为使用了默认交换机。
>
> 交换机定义了路由键（RoutingKey）以及队列（Queue）和交换机之间的绑定关系。生产者在生产消息后，通过指定路由键发送给指定的交换机。交换机再将消息发送到绑定的队列中。
> RabbitMQ 提供了四种交换机：
>
> 1. Default Exchange：默认交换机
> 2. Direct Exchange：直连交换机，根据Routing Key(路由键)进行投递到不同队列。
> 3. Fanout Exchange：扇形交换机，采用广播模式，根据绑定的交换机，路由到与之对应的所有队列。
> 4. Topic Exchange：主题交换机，对路由键进行模式匹配后进行投递，符号#表示一个或多个词，*表示一个词。
> 5. Header Exchange：头交换机，不处理路由键。而是根据发送的消息内容中的headers属性进行匹配。

- **Default Exchange：默认交换机**

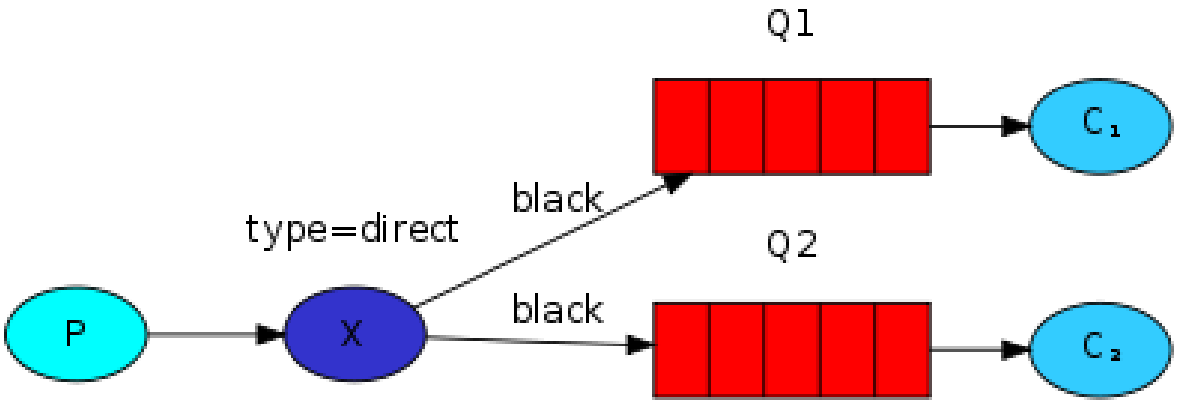


- **Direct Exchange：直连交换机**

> 单个绑定，一个路由键对应一个队列。

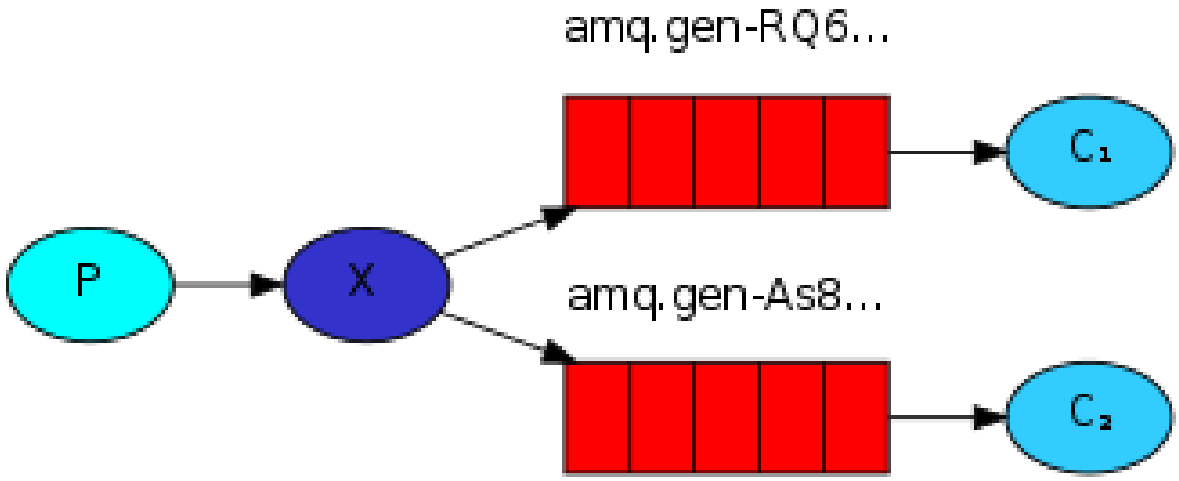


> 多个绑定，一个路由键对应多个队列，消息会分别投递到两个队列中



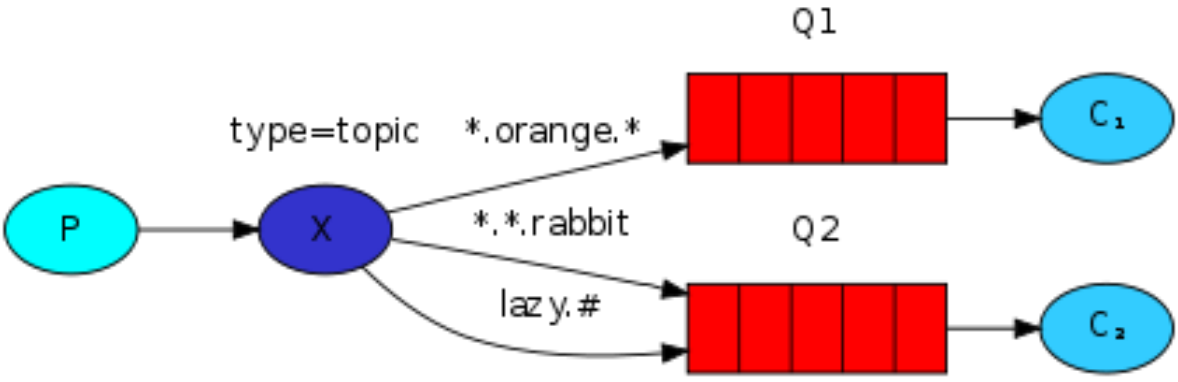
- **Fanout Exchange：扇形交换机**

> 扇形交换机，采用广播模式，根据绑定的交换机，路由到与之对应的所有队列。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。



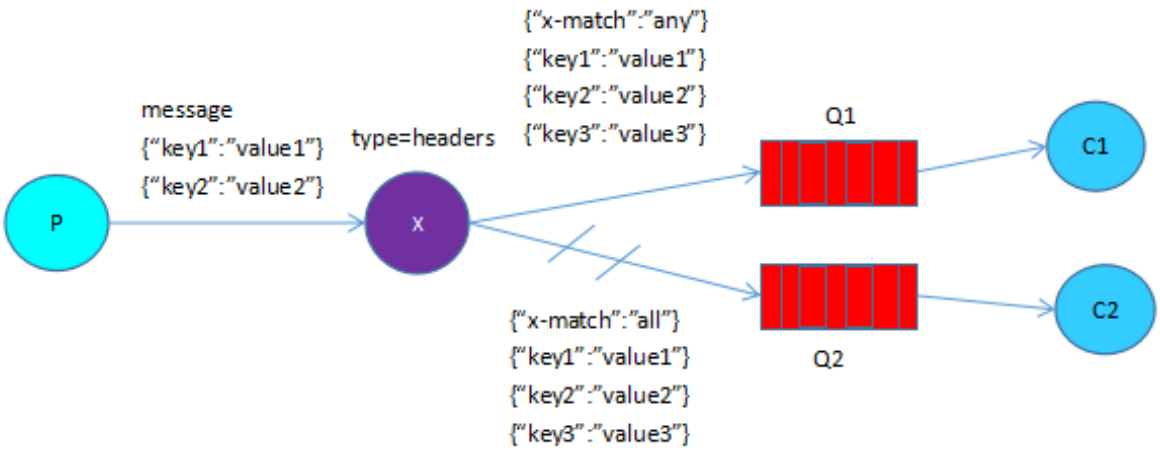
- **Topic Exchange：主题交换机**

> 主题交换机，对路由键进行模式匹配后进行投递，符号#表示一个或多个词，*表示一个词。因此“abc.#”能够匹配到“abc.def.ghi”，但是“abc.”* 只会匹配到“abc.def”。



- **Header Exchange：头交换机**

> 头交换机，不处理路由键。而是根据发送的消息内容中的headers属性进行匹配。在绑定Queue与Exchange时指定一组键值对；当消息发送到RabbitMQ时会取到该消息的headers与Exchange绑定时指定的键值对进行匹配；如果完全匹配则消息会路由到该队列，否则不会路由到该队列。headers属性是一个键值对，可以是Hashtable，键值对的值可以是任何类型。而fanout，direct，topic 的路由键都需要要字符串形式的。
>
> 匹配规则x-match有下列两种类型：
>
> x-match = all ： 表示所有的键值对都匹配才能接受到消息
>
> x-match = any ： 表示只要有键值对匹配就能接受到消息



(二) Fanout Exchange

- 生产者

```
package com.soft.fanout;

import com.rabbitmq.client.BuiltinExchangeType;
import com.rabbitmq.client.Channel;
import com.soft.util.RabbitMqUtil;

import java.util.Scanner;

public class FanoutProducer {
    private static final String EXCHANGE_NAME = "fanout";

    public static void main(String[] args) throws Exception {
        // 1、工具类创建信道
        RabbitMqUtil rabbitMqUtil = new RabbitMqUtil();
        Channel channel = rabbitMqUtil.getChannel();
        // 2、创建交换机，指定交换机名字和类型；类型可以使用枚举类
        channel.exchangeDeclare(EXCHANGE_NAME, BuiltinExchangeType.FANOUT);
        // 定义路由键
        String routingKey = "";

        Scanner scanner = new Scanner(System.in);
        while (true){
            System.out.print("请输入消息：");
            String msg = scanner.next();
            // 3、发送消息，发送给指定路由键的交换机
            channel.basicPublish(EXCHANGE_NAME, routingKey, null,
msg.getBytes("UTF-8"));
```

```
            System.out.println("消息发送成功：" + msg);
        }
    }
}

```

- 消费者1

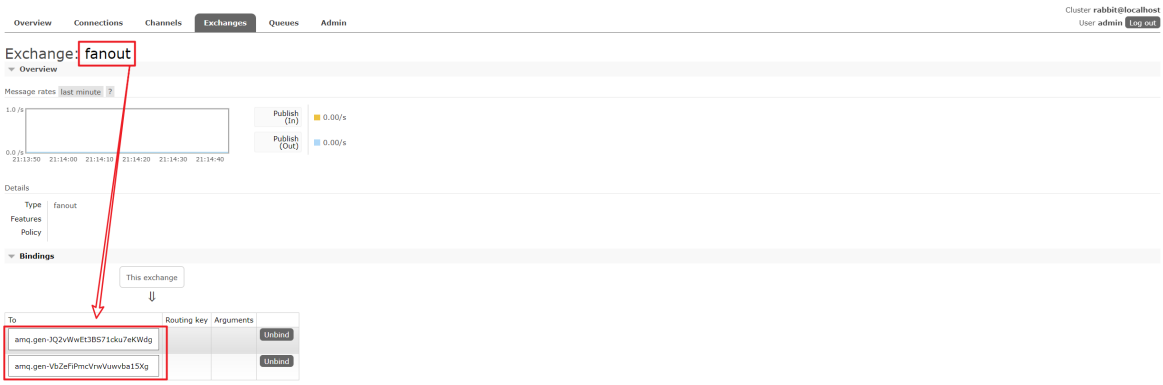
```
package com.soft.fanout;

import com.rabbitmq.client.AMQP;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.DefaultConsumer;
import com.rabbitmq.client.Envelope;
import com.soft.util.RabbitMqUtil;

import java.io.IOException;

public class FanoutConsumer1 {
    private static final String EXCHANGE_NAME = "fanout";

    public static void main(String[] args) throws Exception {
        // 1、调用工具类生成信道
        RabbitMqUtil rabbitMqUtil = new RabbitMqUtil();
        Channel channel = rabbitMqUtil.getChannel();
        // 2、创建临时队列，使用完毕之后会自动删除。getQueue 可以获取队列名称
        String queueName = channel.queueDeclare().getQueue();
        /*
            String queue：队列名字
            String exchange：交换机名字
            String routingKey：路由键
         */
        String routingKey = "";
        // 3、队列绑定交换机
        channel.queueBind(queueName, EXCHANGE_NAME, routingKey);
        /*
            4、接收消息
         */
        DefaultConsumer consumer = new DefaultConsumer(channel){
            @Override
            public void handleDelivery(String consumerTag, Envelope
envelope, AMQP.BasicProperties properties, byte[] body) throws IOException {
                String msg = new String(body);
                System.out.println(consumerTag + "【消费者1接收消息】" + msg);
            }
        };
        System.out.println("消费者1消息接收中...");
        channel.basicConsume(queueName, true, consumer);
    }
}

```

- 消费者2

> 与消费者1类似。

- 结果

> 启动生产者和消费者之后，生产者创建交换机 fanout，指定 RoutingKey。消费者创建信道，并和交换机进行绑定。如图：fanout 交换机下绑定了两个队列。



(三) Direct Exchange

> 与 Fanout Exchange 类似，区别在于 Fanout Exchange 发送的消息每个消费者都能接收；Direct Exchange 发送的消息可以指定消费者接收。

- 生产者

```java
package com.soft.direct;

import com.rabbitmq.client.BuiltinExchangeType;
import com.rabbitmq.client.Channel;
import com.soft.util.RabbitMqUtil;

import java.util.Scanner;

public class DirectProducer {
    private static final String EXCHANGE_NAME = "direct";

    public static void main(String[] args) throws Exception {
        // 1、工具类创建信道
        RabbitMqUtil rabbitMqUtil = new RabbitMqUtil();
        Channel channel = rabbitMqUtil.getChannel();
        // 2、创建交换机，指定交换机名字和类型；类型可以使用枚举类
        channel.exchangeDeclare(EXCHANGE_NAME, BuiltinExchangeType.DIRECT);

        Scanner scanner = new Scanner(System.in);
        while (true){
            System.out.print("请输入消费群体：");
            // 定义路由键
            String routingKey = scanner.next();
            System.out.print("请输入消息：");
            String msg = scanner.next();
            // 3、发送消息，发送给指定路由键的交换机
            channel.basicPublish(EXCHANGE_NAME, routingKey, null,
msg.getBytes("UTF-8"));
            System.out.println("消息发送成功：" + msg);
        }
    }
}
```

- 消费者1

```java
package com.soft.direct;

import com.rabbitmq.client.*;
import com.soft.util.RabbitMqUtil;

import java.io.IOException;
import java.util.Scanner;

public class DirectConsumer1 {
    private static final String EXCHANGE_NAME = "direct";

    public static void main(String[] args) throws Exception {
        // 1、工具类创建信道
        RabbitMqUtil rabbitMqUtil = new RabbitMqUtil();
        Channel channel = rabbitMqUtil.getChannel();

        // 2、创建临时队列，使用完毕之后会自动删除。getQueue 可以获取队列名称
        String queueName = channel.queueDeclare().getQueue();
        /*
            String queue：队列名字
            String exchange：交换机名字
            String routingKey：路由键
         */
        channel.queueBind(queueName, EXCHANGE_NAME, "orange");
        channel.queueBind(queueName, EXCHANGE_NAME, "public");
        /*
            3、接收消息
         */
        DefaultConsumer consumer = new DefaultConsumer(channel){
            @Override
            public void handleDelivery(String consumerTag, Envelope
envelope, AMQP.BasicProperties properties, byte[] body) throws IOException {
                String msg = new String(body);
                System.out.println(consumerTag + "【消费者1接收消息】" + msg);
            }
        };
        System.out.println("消费者1消息接收中...");
        channel.basicConsume(queueName, true, consumer);
    }
}
```

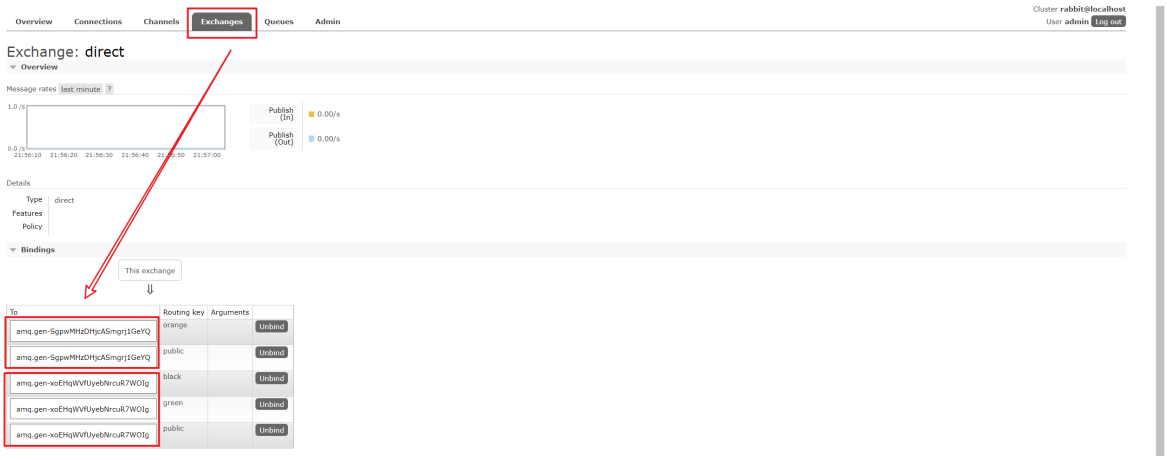
- 消费者2

```java
package com.soft.direct;

import com.rabbitmq.client.AMQP;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.DefaultConsumer;
import com.rabbitmq.client.Envelope;
import com.soft.util.RabbitMqUtil;

import java.io.IOException;

```

```java
public class DirectConsumer2 {
    private static final String EXCHANGE_NAME = "direct";

    public static void main(String[] args) throws Exception {
        // 1、工具类创建信道
        RabbitMqUtil rabbitMqUtil = new RabbitMqUtil();
        Channel channel = rabbitMqUtil.getChannel();

        // 2、创建临时队列，使用完毕之后会自动删除。getQueue 可以获取队列名称
        String queueName = channel.queueDeclare().getQueue();
        /*
            String queue：队列名字
            String exchange：交换机名字
            String routingKey：路由键
         */
        channel.queueBind(queueName, EXCHANGE_NAME, "black");
        channel.queueBind(queueName, EXCHANGE_NAME, "green");
        channel.queueBind(queueName, EXCHANGE_NAME, "public");
        /*
            3、接收消息
         */
        DefaultConsumer consumer = new DefaultConsumer(channel){
            @Override
            public void handleDelivery(String consumerTag, Envelope
envelope, AMQP.BasicProperties properties, byte[] body) throws IOException {
                String msg = new String(body);
                System.out.println(consumerTag + "【消费者2接收消息】" + msg);
            }
        };
        System.out.println("消费者2消息接收中...");
        channel.basicConsume(queueName, true, consumer);
    }
}

```

- 结果

> 交换机和队列的绑定关系如下，direct 交换机绑定了 2 个队列。Tom 和 Jack 分别是两个队列，同时两个队列又绑定了公共的 Routing Key。也就是说 Tom 和 Jack 可以分别接收消息，也可以一起接收消息。



(四) Topics Exchange

- 生产者

```
package com.soft.topics;

import com.rabbitmq.client.BuiltinExchangeType;
import com.rabbitmq.client.Channel;
import com.soft.util.RabbitMqUtil;

import java.util.Scanner;

public class TopicProducer {
    private static final String EXCHANGE_NAME = "topic";

    public static void main(String[] args) throws Exception {
        // 1、工具类创建信道
        RabbitMqUtil rabbitMqUtil = new RabbitMqUtil();
        Channel channel = rabbitMqUtil.getChannel();
        // 2、创建交换机，指定交换机名字和类型；类型可以使用枚举类
        channel.exchangeDeclare(EXCHANGE_NAME, BuiltinExchangeType.TOPIC);

        Scanner scanner = new Scanner(System.in);
        while (true){
            System.out.print("请输入消费群体：");
            // 定义路由键
            String routingKey = scanner.next();
            System.out.print("请输入消息：");
            String msg = scanner.next();
            // 3、发送消息，发送给指定路由键的交换机
            channel.basicPublish(EXCHANGE_NAME, routingKey, null,
msg.getBytes("UTF-8"));
            System.out.println("消息发送成功：" + msg);
        }
    }
}
```

- 消费者1

```
package com.soft.topics;

import com.rabbitmq.client.AMQP;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.DefaultConsumer;
import com.rabbitmq.client.Envelope;
import com.soft.util.RabbitMqUtil;

import java.io.IOException;

public class TopicConsumer1 {
    private static final String EXCHANGE_NAME = "topic";

    public static void main(String[] args) throws Exception {
        // 1、工具类创建信道
```

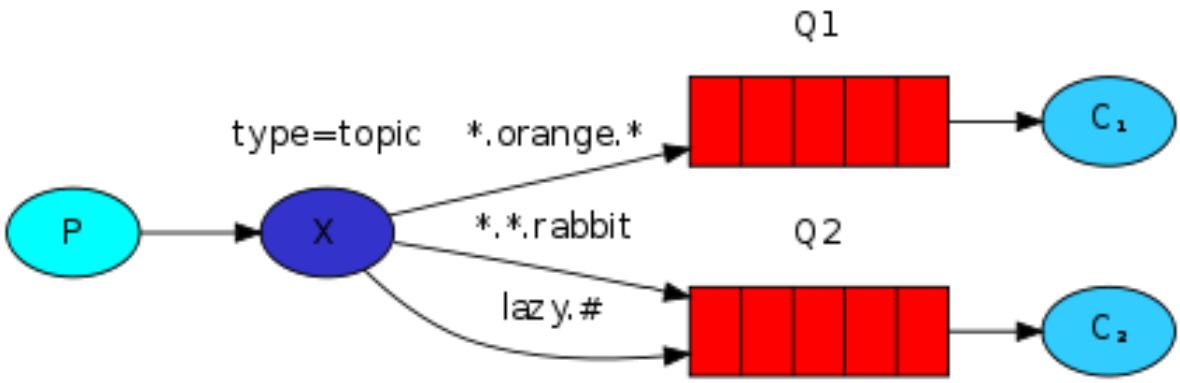
```
        RabbitMqUtil rabbitMqUtil = new RabbitMqUtil();
        Channel channel = rabbitMqUtil.getChannel();

        // 2、创建临时队列，使用完毕之后会自动删除。getQueue 可以获取队列名称
        String queueName = channel.queueDeclare().getQueue();
        /*
            String queue：队列名字
            String exchange：交换机名字
            String routingKey：路由键
         */
        channel.queueBind(queueName, EXCHANGE_NAME, "*.orange.*");
        /*
            3、接收消息
         */
        DefaultConsumer consumer = new DefaultConsumer(channel){
            @Override
            public void handleDelivery(String consumerTag, Envelope
envelope, AMQP.BasicProperties properties, byte[] body) throws IOException {
                String msg = new String(body);
                System.out.println(consumerTag + "【消费者1接收消息】" + msg);
            }
        };
        System.out.println("消费者1消息接收中...");
        channel.basicConsume(queueName, true, consumer);
    }
}
```

- 消费者2

```
package com.soft.topics;

import com.rabbitmq.client.AMQP;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.DefaultConsumer;
import com.rabbitmq.client.Envelope;
import com.soft.util.RabbitMqUtil;

import java.io.IOException;

public class TopicConsumer2 {
    private static final String EXCHANGE_NAME = "topic";

    public static void main(String[] args) throws Exception {
        // 1、工具类创建信道
        RabbitMqUtil rabbitMqUtil = new RabbitMqUtil();
        Channel channel = rabbitMqUtil.getChannel();

        // 2、创建临时队列，使用完毕之后会自动删除。getQueue 可以获取队列名称
        String queueName = channel.queueDeclare().getQueue();
        /*
            String queue：队列名字
            String exchange：交换机名字
            String routingKey：路由键
         */
        channel.queueBind(queueName, EXCHANGE_NAME, "*.*.rabbit");
```

```
27          channel.queueBind(queueName, EXCHANGE_NAME, "lazy.#");
28          /*
29              3、接收消息
30           */
31          DefaultConsumer consumer = new DefaultConsumer(channel){
32              @Override
33              public void handleDelivery(String consumerTag, Envelope
    envelope, AMQP.BasicProperties properties, byte[] body) throws IOException {
34                  String msg = new String(body);
35                  System.out.println(consumerTag + "【消费者2接收消息】" + msg);
36              }
37          };
38          System.out.println("消费者2消息接收中...");
39          channel.basicConsume(queueName, true, consumer);
40      }
41  }
42
```

- 结果



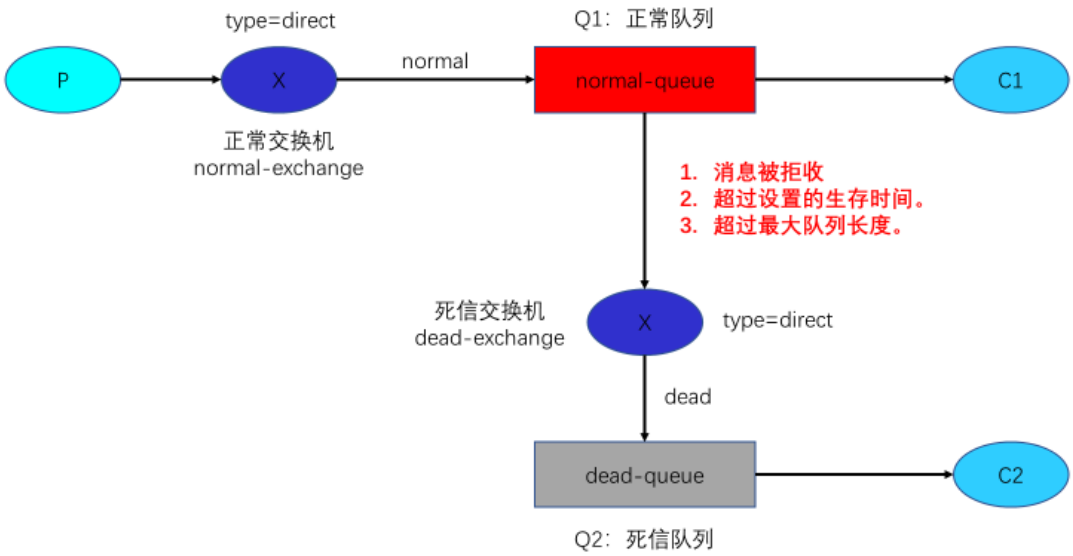
> A message with a routing key set to "quick.orange.rabbit" will be delivered to both queues.
>
> Message "lazy.orange.elephant" also will go to both of them.
>
> On the other hand "quick.orange.fox" will only go to the first queue, and "lazy.brown.fox" only to the second.
>
> "lazy.pink.rabbit" will be delivered to the second queue only once, even though it matches two bindings.
>
> "quick.brown.fox" doesn't match any binding so it will be discarded.
>
> What happens if we break our contract and send a message with one or four words, like "orange" or "quick.orange.male.rabbit"? Well, these messages won't match any bindings and will be lost.
>
> On the other hand "lazy.orange.male.rabbit", even though it has four words, will match the last binding and will be delivered to the second queue.

Routing key	Cumsumer
quick.orange.rabbit	消费者1、消费者2

Routing key	Cumsumer
lazy.orange.elephant	消费者1、消费者2
quick.orange.fox	消费者1
lazy.brown.fox	消费者2
lazy.pink.rabbit	消费者2
quick.brown.fox	丢弃
orange	丢弃
quick.orange.male.rabbit	丢弃
lazy.orange.male.rabbit	消费者2

八、死信队列

> "死信"直观的理解就是死掉的信息。从技术角度看，就是生产者生产的消息，因为某些情况，无法被消费者消费，这样的信息成为死信。RabbitMQ中的信息是存储在队列中，而死信消息存储的队列称为死信队列。
>
> 当你在消费消息时，如果出现以下情况，则消息会进入死信队列。如果配置了死信队列信息，那么该消息将会被丢进死信队列中，如果没有配置，则该消息将会被丢弃。
>
> 1. 消息被拒收，使用 channel.basicNack 或 channel.basicReject ，并且此时requeue 属性被设置为 false 。
> 2. 消息在队列的存活时间超过设置的生存时间（TTL）。
> 3. 消息队列的消息数量已经超过最大队列长度。



（一）超过TTL

- 生产者

> 1. 创建正常交换机，用于传递正常消息
> 2. 创建死信交换机，用于传递死信消息
> 3. 发送消息给正常交换机，路由键是 “normal”
> 4. 发送消息时指定过期时间 10s

```
package com.soft.dead;

import com.rabbitmq.client.AMQP;
import com.rabbitmq.client.BuiltinExchangeType;
import com.rabbitmq.client.Channel;
import com.soft.util.RabbitMqUtil;

public class TTLProducer {
    // 正常交换机名字
    private static final String EXCHANGE_NAME_NORMAL = "exchangeTtlNormal";
    // 死信交换机名字
    private static final String EXCHANGE_NAME_DEAD = "exchangeTtlDead";

    public static void main(String[] args) throws Exception {
        // 1、工具类创建信道
        RabbitMqUtil rabbitMqUtil = new RabbitMqUtil();
        Channel channel = rabbitMqUtil.getChannel();
        // 2、创建正常交换机
        channel.exchangeDeclare(EXCHANGE_NAME_NORMAL,
BuiltinExchangeType.DIRECT);
        // 3、创建死信交换机
        channel.exchangeDeclare(EXCHANGE_NAME_DEAD,
BuiltinExchangeType.DIRECT);
        // 定义路由键
        String routingKey = "OK";

        // 配置消息参数，过期时间单位 ms
        AMQP.BasicProperties props = new
AMQP.BasicProperties().builder().expiration("10000").build();

        for (int i = 0; i < 10; i++) {
            String msg = "MSG" + i;
            // 3、发送消息，发送给指定路由键的交换机
            channel.basicPublish(EXCHANGE_NAME_NORMAL, routingKey, props,
msg.getBytes("UTF-8"));
            System.out.println("消息发送成功：" + msg);
            Thread.sleep(2000);
        }
    }
}
```

- 消费者1（正常队列）

> 1. 创建正常队列，绑定正常交换机，用于接收正常消息，路由键是 “normal”
> 2. 配置正常队列参数，正常队列无法接收到消息时进入死信队列。

```java
package com.soft.dead;

import com.rabbitmq.client.*;
import com.soft.util.RabbitMqUtil;

import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

/**
 * 用于接收正常消息
 */
public class TTLConsumer1 {
    // 正常交换机名字
    private static final String EXCHANGE_NAME_NORMAL = "exchangeTtlNormal";
    // 死信交换机名字
    private static final String EXCHANGE_NAME_DEAD = "exchangeTtlDead";
    // 正常队列名字
    private static final String CHANNEL_NAME_NORMAL = "normal";

    public static void main(String[] args) throws Exception {
        // 1、工具类创建信道
        RabbitMqUtil rabbitMqUtil = new RabbitMqUtil();
        Channel channel = rabbitMqUtil.getChannel();

        // ----------------------------------------------------------------------
        // 2、死信参数配置
        Map<String, Object> arguments = new HashMap<>();
        // 死信交换机名称
        arguments.put("x-dead-letter-exchange", EXCHANGE_NAME_DEAD);
        // 死信交换机路由键
        arguments.put("x-dead-letter-routing-key", "ERR");
        // ----------------------------------------------------------------------

        // 3、创建正常队列
        channel.queueDeclare(CHANNEL_NAME_NORMAL, false, false, false, arguments);
        // 4、绑定正常交换机队列
        channel.queueBind(CHANNEL_NAME_NORMAL, EXCHANGE_NAME_NORMAL, "OK");
        // 5、接收消息
        DefaultConsumer consumer = new DefaultConsumer(channel){
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, byte[] body) throws IOException {
                String msg = new String(body, "UTF-8");
                System.out.println(msg);
            }
        };
        System.out.println("正常队列等待接收消息...");
        channel.basicConsume(CHANNEL_NAME_NORMAL, true, consumer);

    }
}
```

- 消费者2（死信队列）

```java
package com.soft.dead;

import com.rabbitmq.client.AMQP;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.DefaultConsumer;
import com.rabbitmq.client.Envelope;
import com.soft.util.RabbitMqUtil;

import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

/**
 * 用于接收死信消息
 */
public class TTLConsumer2 {
    // 死信交换机名字
    private static final String EXCHANGE_NAME_DEAD = "exchangeTtlDead";
    // 死信队列名字
    private static final String CHANNEL_NAME_DEAD = "dead";

    public static void main(String[] args) throws Exception {
        // 1、工具类创建信道
        RabbitMqUtil rabbitMqUtil = new RabbitMqUtil();
        Channel channel = rabbitMqUtil.getChannel();
        // 2、创建死信队列
        channel.queueDeclare(CHANNEL_NAME_DEAD, false, false, false, null);
        // 3、绑定死信交换机队列
        channel.queueBind(CHANNEL_NAME_DEAD, EXCHANGE_NAME_DEAD, "ERR",
null);
        // 4、接收消息
        DefaultConsumer consumer = new DefaultConsumer(channel){
            @Override
            public void handleDelivery(String consumerTag, Envelope
envelope, AMQP.BasicProperties properties, byte[] body) throws IOException {
                String msg = new String(body, "UTF-8");
                System.out.println(msg);
            }
        };
        System.out.println("死信队列等待接收消息...");
        channel.basicConsume(CHANNEL_NAME_DEAD, true, consumer);

    }
}
```

（二）超过最大长度

> 队列中最多能存储的消息量。

- 生产者

> 1. 创建正常交换机，用于传递正常消息
> 2. 创建死信交换机，用于传递死信消息

> 3. 发送消息给正常交换机，路由键是 “normal”

```
package com.soft.dead;

import com.rabbitmq.client.AMQP;
import com.rabbitmq.client.BuiltinExchangeType;
import com.rabbitmq.client.Channel;
import com.soft.util.RabbitMqUtil;

public class MaxProducer {
    // 正常交换机名字
    private static final String EXCHANGE_NAME_NORMAL = "exchangeTtlNormal";
    // 死信交换机名字
    private static final String EXCHANGE_NAME_DEAD = "exchangeTtlDead";

    public static void main(String[] args) throws Exception {
        // 1、工具类创建信道
        RabbitMqUtil rabbitMqUtil = new RabbitMqUtil();
        Channel channel = rabbitMqUtil.getChannel();
        // 2、创建正常交换机
        channel.exchangeDeclare(EXCHANGE_NAME_NORMAL,
BuiltinExchangeType.DIRECT);
        // 3、创建死信交换机
        channel.exchangeDeclare(EXCHANGE_NAME_DEAD,
BuiltinExchangeType.DIRECT);
        // 定义路由键
        String routingKey = "OK";

        for (int i = 0; i < 10; i++) {
            String msg = "MSG" + i;
            // 3、发送消息，发送给指定路由键的交换机。这里取消TTL案例中的参数配置。
            channel.basicPublish(EXCHANGE_NAME_NORMAL, routingKey, null,
msg.getBytes("UTF-8"));
            System.out.println("消息发送成功：" + msg);
            Thread.sleep(2000);
        }
    }
}
```

- 消费者1（正常队列）

> 1. 创建正常队列，绑定正常交换机，用于接收正常消息，路由键是 “normal”
> 2. 配置正常队列参数，正常队列无法接收到消息时进入死信队列。

注意：测试当前案例时，需要删除之前案例已经创建的队列，否则会创建失败。

```
package com.soft.dead;

import com.rabbitmq.client.*;
import com.soft.util.RabbitMqUtil;

import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

```

```java
/**
 * 用于接收正常消息
 */
public class MaxConsumer1 {
    // 正常交换机名字
    private static final String EXCHANGE_NAME_NORMAL = "exchangeTtlNormal";
    // 死信交换机名字
    private static final String EXCHANGE_NAME_DEAD = "exchangeTtlDead";
    // 正常队列名字
    private static final String CHANNEL_NAME_NORMAL = "normal";

    public static void main(String[] args) throws Exception {
        // 1、工具类创建信道
        RabbitMqUtil rabbitMqUtil = new RabbitMqUtil();
        Channel channel = rabbitMqUtil.getChannel();

        // ----------------------------------------------------------------
        // 2、死信参数配置
        Map<String, Object> arguments = new HashMap<>();
        // 死信交换机名称
        arguments.put("x-dead-letter-exchange", EXCHANGE_NAME_DEAD);
        // 死信交换机路由键
        arguments.put("x-dead-letter-routing-key", "ERR");
        // 正常交换机最大数据量
        arguments.put("x-max-length", 6);
        // ----------------------------------------------------------------

        // 3、创建正常队列
        channel.queueDeclare(CHANNEL_NAME_NORMAL, false, false, false, arguments);
        // 4、绑定正常交换机队列
        channel.queueBind(CHANNEL_NAME_NORMAL, EXCHANGE_NAME_NORMAL, "OK");
        // 5、接收消息
        DefaultConsumer consumer = new DefaultConsumer(channel){
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, byte[] body) throws IOException {
                String msg = new String(body, "UTF-8");
                System.out.println(msg);
            }
        };
        System.out.println("正常队列等待接收消息...");
        channel.basicConsume(CHANNEL_NAME_NORMAL, true, consumer);

    }
}
```

- 消费者2（死信队列）

> 同【TTL】案例

（三）拒收

- 生产者

> 同【超过最大长度】案例

- 消费者1

```
package com.soft.dead;

import com.rabbitmq.client.*;
import com.soft.util.RabbitMqUtil;

import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

/**
 * 用于接收正常消息
 */
public class TTLConsumer1 {
    // 正常交换机名字
    private static final String EXCHANGE_NAME_NORMAL = "exchangeTtlNormal";
    // 死信交换机名字
    private static final String EXCHANGE_NAME_DEAD = "exchangeTtlDead";
    // 正常队列名字
    private static final String CHANNEL_NAME_NORMAL = "normal";

    public static void main(String[] args) throws Exception {
        // 1、工具类创建信道
        RabbitMqUtil rabbitMqUtil = new RabbitMqUtil();
        Channel channel = rabbitMqUtil.getChannel();
        // 2、参数配置
        Map<String, Object> arguments = new HashMap<>();
        // 死信交换机名称
        arguments.put("x-dead-letter-exchange", EXCHANGE_NAME_DEAD);
        // 死信交换机路由键
        arguments.put("x-dead-letter-routing-key", "ERR");
        // 正常交换机最大数据量
        // arguments.put("x-max-length", 6);
        // 3、创建正常队列
        channel.queueDeclare(CHANNEL_NAME_NORMAL, false, false, false,
arguments);
        // 4、绑定正常交换机队列
        channel.queueBind(CHANNEL_NAME_NORMAL, EXCHANGE_NAME_NORMAL, "OK");
        // 5、接收消息
        DefaultConsumer consumer = new DefaultConsumer(channel){
            @Override
            public void handleDelivery(String consumerTag, Envelope
envelope, AMQP.BasicProperties properties, byte[] body) throws IOException {
                String msg = new String(body, "UTF-8");
                if(msg.contains("6")){

                    System.out.println("消息已经发送，但被对方拒收。");
                    // 拒绝消息
                    channel.basicReject(envelope.getDeliveryTag(), false);
```

```
                } else {
                    System.out.println(msg);
                    // 配置手动应答
                    channel.basicAck(envelope.getDeliveryTag(), false);
                }
            }
        };
        System.out.println("正常队列等待接收消息...");
        // 关闭自动应答
        channel.basicConsume(CHANNEL_NAME_NORMAL, false, consumer);
    }
}
```

- 消费者2

> 同【超过最大长度】案例

九、SpringBoot 整合 RabbitMQ

（一）基本配置

1、导入RabbitMQ 依赖

```
<!-- 导入 RabbitMQ 依赖 -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

2、配置服务连接

```
spring:
  rabbitmq:
    host: 192.168.88.88
    port: 5672
    username: admin
    password: 123456
```

（二）简单队列模式

1、编写配置类，创建队列

```java
package com.soft.config;

import org.springframework.amqp.core.Queue;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class HelloConfiguration {
    @Bean
    public Queue helloQueue(){
        return new Queue("hello");
    }
}
```

2、编写生产者和消费者

```java
package com.soft.controller;

import org.springframework.amqp.rabbit.annotation.RabbitHandler;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

import javax.annotation.Resource;

@Controller
@RequestMapping("/hello")
public class HelloController {
    @Resource
    RabbitTemplate rabbitTemplate;

    /**
     * 创建消息
     * @return
     */
    @GetMapping("/p")
    @ResponseBody
    public String producer(){
        rabbitTemplate.convertAndSend("hello", "简单队列模式！");
        return "消息发送成功！";
    }

    /**
     * 消费消息
     * @param msg
     * @return
     */
    // 监听队列
    @RabbitListener(queues = {"hello"})
    // MQ 处理器
    @RabbitHandler
```

```
    public void consumer(String msg){
        System.out.println("【接收到消息】" + msg);
    }
}
```

（三）工作队列模式

1、编写配置类，创建队列

```
package com.soft.config;

import org.springframework.amqp.core.Queue;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class WorkConfiguration {
    @Bean
    public Queue workQueue(){
        return new Queue("work");
    }
}
```

2、编写生产者和消费者

```
package com.soft.controller;

import org.springframework.amqp.rabbit.annotation.RabbitHandler;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

import javax.annotation.Resource;

@Controller
@RequestMapping("/work")
public class WorkController {
    @Resource
    RabbitTemplate rabbitTemplate;

    /**
     * 创建消息
     * @return
     */
    @GetMapping("/p")
    @ResponseBody
    public String producer(){
        rabbitTemplate.convertAndSend("work", "工作队列模式！");
        return "消息发送成功！";
    }

```

```
    /**
     * 消费消息
     * @param msg
     * @return
     */
    // 监听队列
    @RabbitListener(queues = {"work"})
    // MQ 处理器
    @RabbitHandler
    public void consumer1(String msg){
        System.out.println("【消费者1，接收到消息】" + msg);
    }
    /**
     * 消费消息
     * @param msg
     * @return
     */
    // 监听队列
    @RabbitListener(queues = {"work"})
    // MQ 处理器
    @RabbitHandler
    public void consumer2(String msg){
        System.out.println("【消费者2，接收到消息】" + msg);
    }
}
```

(四) Direct Exchange

1、编写配置类，创建交换机和队列

```
package com.soft.config;

import org.springframework.amqp.core.Binding;
import org.springframework.amqp.core.BindingBuilder;
import org.springframework.amqp.core.DirectExchange;
import org.springframework.amqp.core.Queue;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class DirectConfiguration {
    @Bean
    public Queue queue1(){
        return new Queue("Q1");
    }
    @Bean
    public Queue queue2(){
        return new Queue("Q2");
    }

    @Bean
    public DirectExchange directExchange(){
        return new DirectExchange("direct.exchange");
    }

```

```
    @Bean
    public Binding binding1(){
        return BindingBuilder.bind(queue1()).to(directExchange()).with("orange");
    }
    @Bean
    public Binding binding2(){
        return BindingBuilder.bind(queue2()).to(directExchange()).with("black");
    }
    @Bean
    public Binding binding3(){
        return BindingBuilder.bind(queue2()).to(directExchange()).with("green");
    }
}
```

2、编写生产者和消费者

```
package com.soft.controller;

import org.springframework.amqp.rabbit.annotation.RabbitHandler;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

import javax.annotation.Resource;
import java.io.IOException;

@Controller
@RequestMapping("/direct")
public class DirectController {
    @Resource
    RabbitTemplate rabbitTemplate;

    /**
     * 创建消息
     * @return
     */
    @GetMapping("/p")
    @ResponseBody
    public String producer(String routingKey, String msg){
        rabbitTemplate.convertAndSend("direct.exchange", routingKey, msg);
        return "消息发送成功！";
    }

    /**
     * 消费消息
     * @param msg
     * @return
     */
    // 监听队列
```

```
37      @RabbitListener(queues = {"Q1"})
38      // MQ 处理器
39      @RabbitHandler
40      public void consumer1(String msg) throws IOException {
41          System.out.println("【消费者1，接收到消息】" + msg);
42      }
43      /**
44       * 消费消息
45       * @param msg
46       * @return
47       */
48      // 监听队列
49      @RabbitListener(queues = {"Q2"})
50      // MQ 处理器
51      @RabbitHandler
52      public void consumer2(String msg) throws IOException {
53          System.out.println("【消费者2，接收到消息】" + msg);
54      }
55  }
```