

## 一、MQ介绍

- 1、什么是MQ? 为什么要用MQ?
- 2、MQ的优缺点
- 3、几大主流MQ产品特点比较
- 4、关于RabbitMQ

## 二、Rabbitmq快速上手

- 1、版本选择
- 2、安装RabbitMQ服务
- 3、启用RabbitMQ管理插件
- 4、理解Exchange和Queue
- 5、理解Connection和Channel
- 6、RabbitMQ中的核心概念总结

## 三、章节总结

# RabbitMQ快速实战以及核心概念详解

--楼兰

## 一、MQ介绍

### 1、什么是MQ? 为什么要用MQ?

ChatGPT中对于消息队列的介绍是这样的:

什么是消息队列

消息队列是一种在应用程序之间传递消息的技术。它提供了一种异步通信模式,允许应用程序在不同的时间处理消息。消息队列通常用于解耦应用程序,以便它们可以独立地扩展和修改。在消息队列中,消息发送者将消息发送到队列中,然后消息接收者从队列中接收消息。这种模式允许消息接收者按照自己的节奏处理消息,而不必等待消息发送者处理完消息。常见的消息队列包括RabbitMQ、Kafka和ActiveMQ等。

MQ: MessageQueue, 消息队列。这东西分两个部分来理解: 队列, 是一种FIFO 先进先出的数据结构。消息: 在不同应用程序之间传递的数据。**将消息以队列的形式存储起来, 并且在不同的应用程序之间进行传递, 这就成了MessageQueue。**而MQ的作用, 从刚才ChatGPT的介绍中就能够抽象出三个关键字: 异步、解耦、削峰。但是这什么意思呢? 跟开发有什么关系? 我们从一个简单的SpringBoot应用开始说起。

首先搭建一个普通的Maven项目, 在pom.xml中引入SpringBoot的依赖:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-dependencies</artifactId>
      <version>2.4.5</version>
      <type>pom</type>
      <scope>import</scope>
```

```

        </dependency>
    </dependencies>
</dependencyManagement>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
    </dependency>
</dependencies>

```

然后，添加一个监听器类

```

public class MyApplicationListener implements ApplicationListener<ApplicationEvent>
{
    @Override
    public void onApplicationEvent(ApplicationEvent applicationEvent) {
        System.out.println("=====> MyApplicationListener: "+applicationEvent);
    }
}

```

接下来，添加一个SpringBoot启动类。在启动类中加入自己的这个监听器。

```

@SpringBootApplication
public class AppDemo implements CommandLineRunner {
    public static void main(String[] args) {
        SpringApplication application = new SpringApplication(AppDemo.class);
        application.addListeners(new MyApplicationListener());
        application.run(args);
    }

    @Resource
    private ApplicationContext applicationContext;
    @Override
    public void run(String... args) throws Exception {
        applicationContext.publishEvent(new ApplicationEvent("myEvent"){
        });
    }
}

```

好了。不用添加配置文件，直接启动就行。然后可以看到这样的结果：

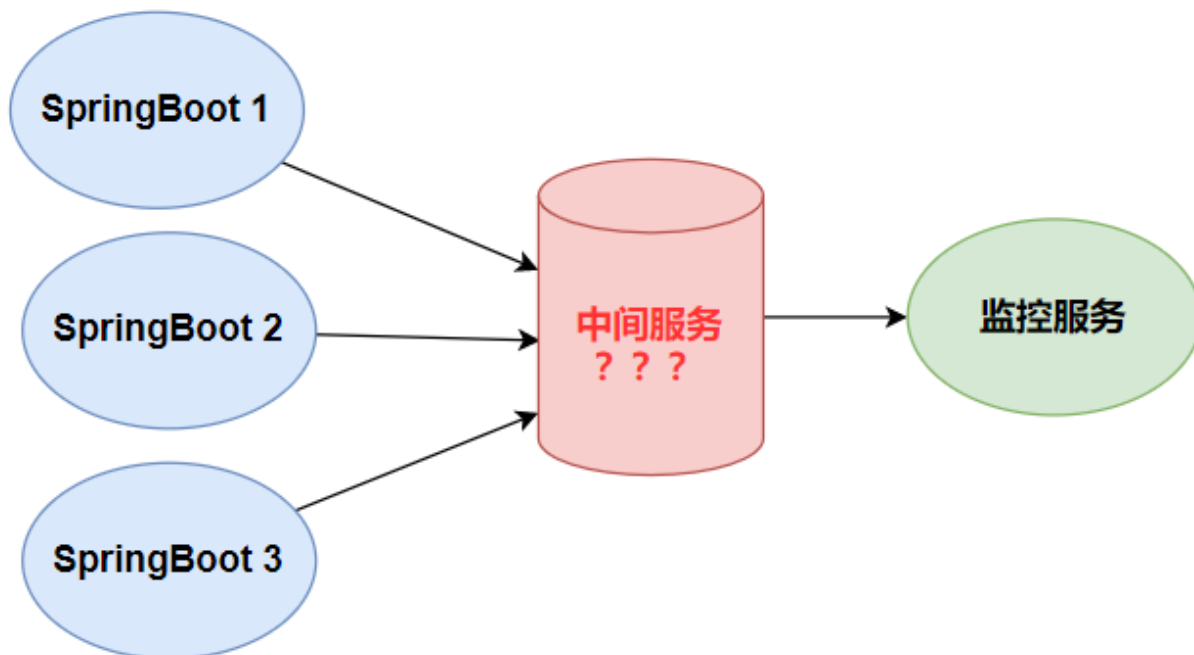
```
D:\dev-hook\Java\jdk1.8.0_45\bin\java.exe ...  
=====> MyApplicationListener: org.springframework.boot.context.event.ApplicationStartingEvent[source=org.springframework.boot.SpringApplication@60c6f5b]  
=====> MyApplicationListener: org.springframework.boot.context.event.ApplicationEnvironmentPreparedEvent[source=org.springframework.boot.SpringApplication@60c6f5b]  
  
      _ ____  
     /   \__  ___(C)____ _ __VVVVV  
    / __  |___/_|_|_|_|_|_|_|_|VVVVV  
   / ____|_|_|_|_|_|_|_|_|_|_|)|)  
  / _____|_|_|_|_|_|_|_|_|_|_|/  
=====|_|=====|_|_|_|_|_|_|_  
:: Spring Boot ::                (v2.4.5)  
  
=====> MyApplicationListener: org.springframework.boot.context.event.ApplicationContextInitializedEvent[source=org.springframework.boot.SpringApplication@60c6f5b]  
2023-03-24 09:37:22.694 INFO 17744 --- [main] com.rooy.AppDemo : Starting AppDemo using Java 1.8.0_45 on USER-20221017CE with PID 17744 (E:\a  
2023-03-24 09:37:22.696 INFO 17744 --- [main] com.rooy.AppDemo : No active profile set, falling back to default profiles: default  
=====> MyApplicationListener: org.springframework.boot.context.event.ApplicationPreparedEvent[source=org.springframework.boot.SpringApplication@60c6f5b]  
=====> MyApplicationListener: org.springframework.boot.context.event.ContextRefreshedEvent[source=org.springframework.boot.context.annotation.AnnotationConfigApplica  
2023-03-24 09:37:23.051 INFO 17744 --- [main] com.rooy.AppDemo : Started AppDemo in 0.625 seconds (JVM running for 1.381)  
=====> MyApplicationListener: org.springframework.boot.context.event.ApplicationStartedEvent[source=org.springframework.boot.SpringApplication@60c6f5b]  
=====> MyApplicationListener: org.springframework.boot.availability.AvailabilityChangeEvent[source=org.springframework.context.annotation.AnnotationConfigApplicationConte  
=====> MyApplicationListener: com.rooy.AppDemo$1[source=myEvent] : 自己发布的事件  
=====> MyApplicationListener: org.springframework.boot.availability.AvailabilityReadyEvent[source=org.springframework.boot.SpringApplication@60c6f5b]  
=====> MyApplicationListener: org.springframework.boot.availability.AvailabilityChangeEvent[source=org.springframework.context.annotation.AnnotationConfigApplicationConte  
=====> MyApplicationListener: org.springframework.context.event.ContextClosedEvent[source=org.springframework.context.annotation.AnnotationConfigApplicationConte@55b699ef, sta  
  
Process finished with exit code 0
```

扩展问题：MyApplicationListener使用@Component注解加入Spring容器和这样手动添加监听有什么区别？

从这个例子中可以看到，在run方法中，我们使用applicationContext发布了一个myEvent事件，然后通过自定义的监听器MyApplicationListener，就监听了到这个myEvent事件。这个过程中，applicationContext担任了发布消息的功能，称为消息生产者Producer，而MyApplicationListener担任了消费处理这个消息的功能，称为消息消费者Consumer。Producer和Consumer他们的运行状况互不干涉，没有Consumer，Producer一样正常运行，反过来也一样。也就是说，推送Producer和Consumer正常工作的，只有发布的这些事件。这种方式就称为事件驱动。

从这个简单的例子中你可以看到，SpringBoot内部就是集成了事件驱动机制的。SpringBoot会将自己应用过程中发生的每一个重要的运行步骤都通过事件发送出来。你会发现这些事件对于监控一个SpringBoot应用挺有用的。如果我想要另外搭建一个监控系统，也像MyApplicationListener一样监听SpringBoot的这些事件，应该要怎么做呢？

直接监听肯定是不行的，因为SpringBoot中的这些事件只在应用内部有效。因此，需要独立出一个中间服务，这样才可以去统一接收SpringBoot应用的这些事件。



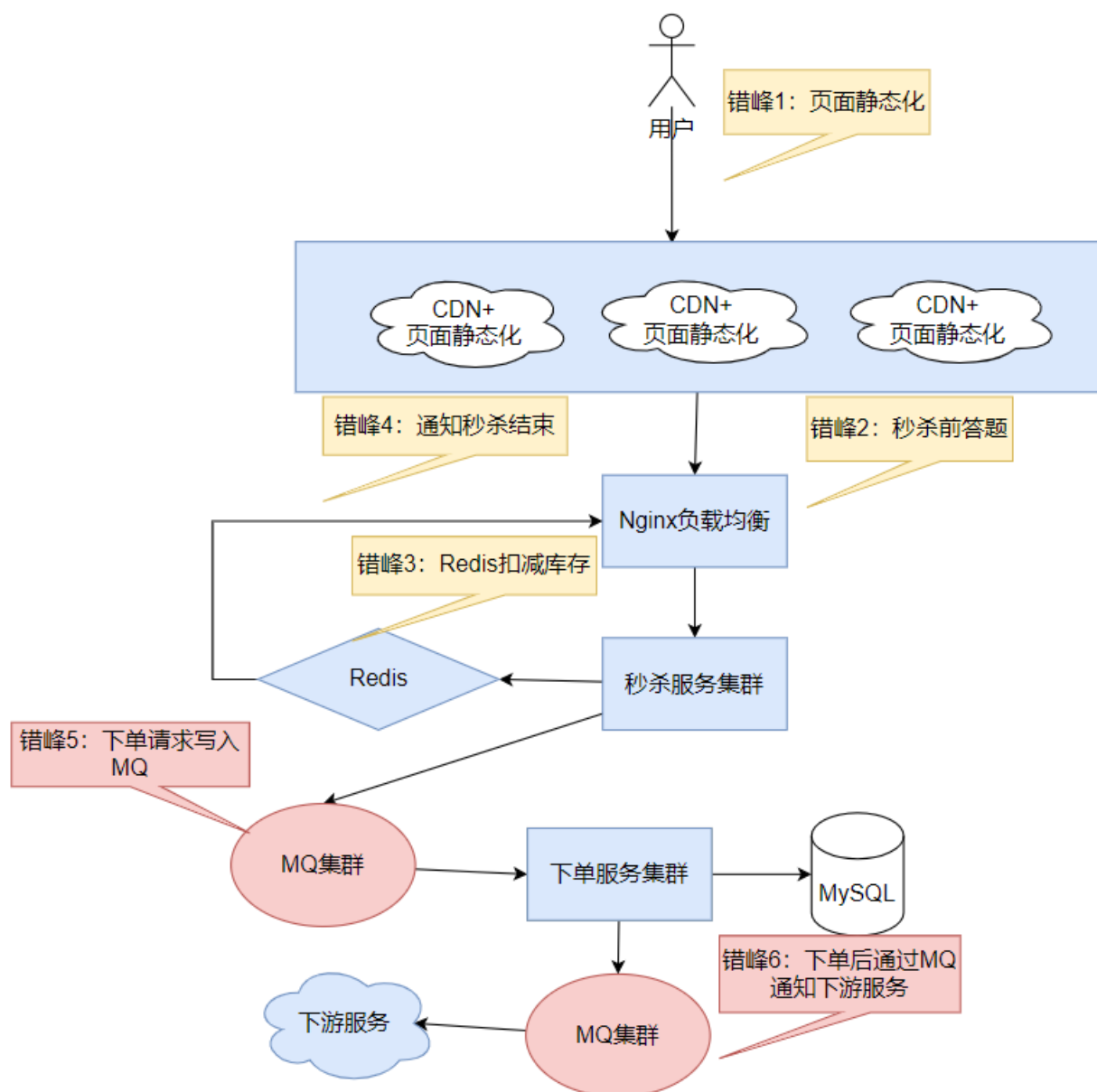
这时这个神秘的中间服务要保证这些系统可以正常工作，应该要有哪些特性呢？

1、SpringBoot应用和监控服务应该是**解耦**的。不管有没有监控服务，SpringBoot应用都要是可以正常运行的。同时，不管监控服务是用什么语言开发的，同样不应该影响SpringBoot应用的正常运行。更进一步，不管监控服务要部署多少个，同样也不应该影响SpringBoot应用的正常运行。反过来，从监控服务看SpringBoot应用也应该是一样的。这就需要这个中间服务可以提供不同语言的客户端，通过不同客户端让消息生产者和消息消费者之间彻底解耦。

2、SpringBoot应用和监控服务之间处理消息应该是**异步**的。基于解耦的关系，SpringBoot应用并不需要知道监控服务有没有运行。所以他并不需要将消息直接发送到监控服务，也不需要保证消息一定会被监控服务处理。他只要将消息发到中间服务就可以了。而监控服务可以在SpringBoot应用发布了时间之后，随时去接收处理这些消息。

3、这个中间服务需要可以协调双方的事件处理速度，产生**削峰填谷**的效果。监控服务一般都是希望每五分钟接收到SpringBoot发布过来的消息，然后进行一次统计，但是SpringBoot应用发布的事件频率却是不确定的。如果SpringBoot应用在五分钟内产生了海量的消息，就有可能让监控服务内存撑爆，处理不过来。而监控服务如果加大内存，SpringBoot应用又有可能在五分钟内根本没有消息，监控服务的内存就白加了。监控服务的内存配大配小都不合适。这时候，就需要这个中间服务能够将这些消息暂存起来，让监控服务可以按照自己的能力慢慢处理问题。这就是中间服务的削峰填谷的作用。

而MQ也就是为了这样的场景创建的中间服务。MQ中间件在很多业务场景中都扮演着很重要的角色。例如下图是一个典型的秒杀场景业务图：



在后面的实战项目中，会带大家从头到尾搭建一个这样的秒杀系统。在这其中，对于后端最重要的优化就是使用MQ。在典型的秒杀场景，瞬间产生的大量下单请求很容易让后端的下单服务崩溃。这时，就可以让下单系统将订单消息发送到MQ中间暂存起来，而后端的额下单服务就可以从MQ中获取数据，按照自己的处理能力，慢慢进行下单。另外，下单是一个比较复杂的业务，需要通知支付系统、库存系统、物流系统、营销系统等大量的下游系统。下单系统光一个个通知这些系统，就会需要很长时间。这时，就可以将下单完成的消息发送到MQ，然后下游的各种系统可以从MQ中获取下单完成的消息，进行异步处理。这样也能极大提高下单系统的性能。

## 2、MQ的优缺点

这时候你可能在想，SpringBoot已经提供了本地的事件驱动支持。那么我是不是给SpringBoot应用加上一些web接口，基于这些web接口不就可以将本地的这些系统事件以及自己产生的这些事件往外部应用推送，那这不就成了一个MQ服务了吗？单其实上面列出了MQ的的很多优点。但是在具体使用MQ时，也会带来很多的缺点：

- 系统可用性降低

系统引入的外部依赖增多，系统的稳定性就会变差。一旦MQ宕机，对业务会产生影响。这就需要考虑如何保证MQ的高可用。

- 系统复杂度提高

引入MQ后系统的复杂度会大大提高。以前服务之间可以进行同步的服务调用，引入MQ后，会变为异步调用，数据的链路就会变得更复杂。并且还会带来其他一些问题。比如：消息如何高效存储、如何定期维护、如何监控、如何溯源等等。如何保证消费不会丢失？不会被重复调用？怎么保证消息的顺序性等问题。

- 消息安全性问题

引入MQ后，消息需要在MQ中存储起来。这时就会带来很多网络造成的数据安全问题。比如如何快速保存海量消息？如何保证消息不丢失？不被重复处理？怎么保证消息的顺序性？如何保证消息事务完整等问题。

所以MQ的应用场景虽然比较简单，但是随着深度分析业务场景，也会随之产生非常多的问题。甚至为此指定的业务标准都出现过好几套，比如JMX，AMQP等。因此，这才需要构建出RabbitMQ等这些中间件，来完整的处理MQ遇到的这些问题。

### 3、几大主流MQ产品特点比较

所以MQ通常用起来比较简单，但是实现上是非常复杂的。基本上MQ代表了业界高可用、高并发、高可扩展三高架构的所有设计精髓。在MQ长期发展过程中，诞生了很多MQ产品，但是有很多MQ产品都已经逐渐被淘汰了。比如早期的ZeroMQ,ActiveMQ等。目前最常用的MQ产品包括Kafka、RabbitMQ和RocketMQ。我们对这三个产品做下简单的比较，重点需要理解他们的适用场景。

	优点	缺点	使用场景
kafka	吞吐量非常大，性能非常好，集群高可用。	会丢数据，功能比较单一。	日志分析，大数据采集
RabbitMQ	消息可靠性高，功能全面。	吞吐量比较低，消息积累会影响性能，erlang语言不好定制。	小规模场景
RocketMQ	高吞吐，高性能，高可用，功能全面。	开源版功能不如云上版，官方文档比较简单，客户端只支持java。	几乎全场景

这里的优缺点并不是绝对的，因为每个产品都在不断演进。比如Kafka现在基本可以做到数据不丢失。RabbitMQ的Stream队列就是模拟Kafka的实现机制，消息吞吐量也提升了非常多。另外也还有很多新的MQ产品体现了更强大的竞争力，比如Pulsar。

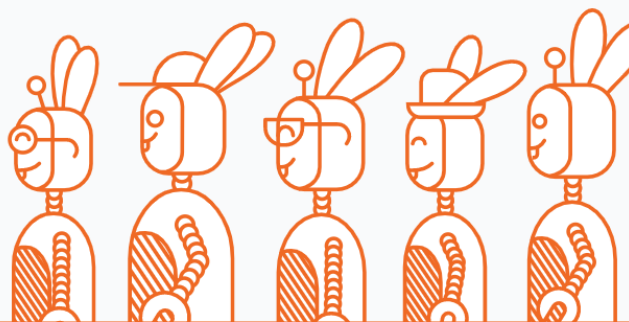
### 4、关于RabbitMQ

RabbitMQ的历史可以追溯到2006年，是一个非常老牌的MQ产品，使用非常广泛。同时期的很多MQ产品都已经逐渐被业界淘汰了，比如2003年诞生的ActiveMQ，2012年诞生的ZeroMQ，但是RabbitMQ却依然稳稳占据一席之地，足可见他的经典。官网地址 <https://www.rabbitmq.com/>。

## Quorum queues

A webinar on high availability and data safety in messaging

[Learn more](#)



RabbitMQ is the most widely deployed open source message broker.

### Updates

1. [RabbitMQ 3.11.10](#) 02 Mar 2023
2. [RabbitMQ 3.10.19](#) 02 Mar 2023

RabbitMQ虽然是开源的，但是是基于erlang语言开发的。这个语言比较小众，用得不是很多。因此很少研究源码。

RabbitMQ的应用相当广泛，拥有很多非常强大的特性：

## OSS RabbitMQ Features



### Asynchronous Messaging

Supports [multiple messaging protocols](#), [message queuing](#), [delivery acknowledgement](#), [flexible routing to queues](#), [multiple exchange type](#).



### Developer Experience

Deploy with [Kubernetes](#), [BOSH](#), [Chef](#), [Docker](#) and [Puppet](#). Develop cross-language messaging with favorite programming languages such as: Java, .NET, PHP, Python, JavaScript, Ruby, Go, [and many others](#).



### Distributed Deployment

Deploy as [clusters](#) for high availability and throughput; [federate](#) across multiple availability zones and regions.



### Enterprise & Cloud Ready

Pluggable [authentication](#), [authorisation](#), supports [TLS](#) and [LDAP](#). Lightweight and easy to deploy in public and private clouds.



### Tools & Plugins

Diverse array of [tools and plugins](#) supporting continuous integration, operational metrics, and integration to other enterprise systems. Flexible [plug-in approach](#) for extending RabbitMQ functionality.



### Management & Monitoring

HTTP-API, command line tool, and UI for [managing and monitoring](#) RabbitMQ.

Asynchronous Message(异步消息)、Developer Experience(开发体验)、Distributed Deployment(分布式部署)、Enterprise & Cloud Ready(企业云部署)、Tools & Plugins(工具和插件)、Management & Monitoring(管理和监控)六大部分

## 二、Rabbitmq快速上手



# 1、版本选择

RabbitMQ版本，通常与他的大的功能是有关系的。3.8.x版本主要是围绕Quorum Queue功能，而3.9.x版本主要是围绕Streams功能。后面的3.10.x版本和3.11.x版本，没有太多新功能，主要是对Quorum Queue和Stream功能做一些修复以及增强，另外，开始增加了一些功能插件，比如OAuth2，MQTT。我们这次就选择目前最新的3.11.10版本。

RabbitMQ是基于Erlang语言开发，所以安装前需要安装Erlang语言环境。需要注意下的是RabbitMQ与Erlang是有版本对应关系的。3.11.10版本的RabbitMQ只支持25.0以上到25.2版本的Erlang。这里要注意一下，如果使用CentOS搭建RabbitMQ服务，Erlang25.0~25.2这几个版本建议的CentOS版本要升级到CentOS8或者CentOS9。

## 2、安装RabbitMQ服务

接下来准备一台CentOS9服务器，快速搭建一个RabbitMQ服务。

RabbitMQ服务安装的方式很多，现在企业级服务多偏向于使用Docker安装。Duchub上已经上传了当前版本的RabbitMQ镜像。用以下docker指令可以安装。

```
docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:3.11-management
```

但是这种方式相当于是官方将一个已经搭建好的环境给你用。这样不便于了解RabbitMQ的一些细节。因此我们这里会采用服务器直接安装的方式。

### 安装RabbitMQ之前需要先安装Erlang语言包

Linux上的安装Erlang稍微有点复杂，需要有非常多的依赖包。简单起见，可以下载rabbitmq提供的zero dependency版本。下载地址 <https://github.com/rabbitmq/erlang-rpm/releases>

下载完成后，可以尝试使用下面的指令安装

```
[root@worker1 tools]# rpm -ivh erlang-25.2.2-1.el9.x86_64.rpm
警告: erlang-25.2.2-1.el9.x86_64.rpm: 头V4 RSA/SHA256 Signature, 密钥 ID 6026dfca:
NOKEY
Verifying...                               ##### [100%]
准备中...                                  ##### [100%]
正在升级/安装...
 1:erlang-25.2.2-1.el9                      ##### [100%]
```

这样Erlang语言包就安装完成了。安装完后可以使用 `erl -version` 指令检测下erlang是否安装成功。

```
[root@worker1 tools]# erl -version
Erlang (SMP,ASYNC_THREADS) (BEAM) emulator version 13.1.4
```

### 安装RabbitMQ

RabbitMQ的安装方式有很多，我们采用RPM安装包的方式。安装包可以到github仓库中下载发布包。下载地址: <https://github.com/rabbitmq/rabbitmq-server/releases>。这里我们下载无依赖版本：  
rabbitmq-server-3.11.10-1.el8.noarch.rpm



```
[root@worker1 tools]# rpm -ivh rabbitmq-server-3.11.10-1.el8.noarch.rpm
警告: rabbitmq-server-3.11.10-1.el8.noarch.rpm: 头V4 RSA/SHA512 Signature, 密钥 ID
6026dfca: NOKEY
Verifying... ##### [100%]
准备中... ##### [100%]
正在升级/安装...
  1:rabbitmq-server-3.11.10-1.el8 ##### [100%]
/usr/lib/tmpfiles.d/rabbitmq-server.conf:1: Line references path below legacy
directory /var/run/, updating /var/run/rabbitmq -> /run/rabbitmq; please update the
tmpfiles.d/ drop-in file accordingly.
```

安装过程中有可能会出现问题需要socat的报错。这时可以直接采用yum方式安装socat依赖。在使用yum时，可以做一个小配置，将yum源配置成阿里的yum源，这样速度会比较快。

```
mv /etc/yum.repos.d/CentOS-Base.repo /etc/yum.repos.d/CentOS-Base.repo.backup

curl -o /etc/yum.repos.d/CentOS-Base.repo
http://mirrors.aliyun.com/repo/Centos-7.repo

yum makecache

然后安装socat
yum install socat
```

安装完成后，可以使用常见的指令维护RabbitMQ的服务。

service rabbitmq-server start --启动Rabbitmq服务。启动应用之前要先启动服务。

rabbitmq-server -deched --后台启动RabbitMQ应用

rabbitmqctl start\_app --启动Rabbitmq

rabbitmqctl stop --关闭Rabbitmq

rabbitmqctl status -- 查看RabbitMQ服务状态。

出现Status为Runtime表示启动成功。

另外，RabbitMQ在Windows上也可以安装。安装方式比较简单，都是几个EXE安装文件，双击安装即可。同样是先安装erlang再安装RabbitMQ。感兴趣朋友也可以自行安装一下，这里就不多说了。

### 3、启用RabbitMQ管理插件

RabbitMQ提供了管理插件，可以快速使用RabbitMQ。在使用之前，需要先打开他的Web管理插件。

```
[root@worker1 ~]# rabbitmq-plugins enable rabbitmq_management
Enabling plugins on node rabbit@worker1:
rabbitmq_management
The following plugins have been configured:
  rabbitmq_management
  rabbitmq_management_agent
  rabbitmq_web_dispatch
Applying plugin configuration to rabbit@worker1...
```

The following plugins have been enabled:

```
rabbitmq_management  
rabbitmq_management_agent  
rabbitmq_web_dispatch
```

started 3 plugins.

插件激活后，就可以访问RabbitMQ的Web控制台了。访问端口15672。RabbitMQ提供的默认用户是guest，密码guest。

但是注意下，默认情况下，只允许在localhost本地登录，远程访问是无法登录的。



Login failed

Username:  \*

Password:  \*

User can only log in via localhost

这时，通常都会创建一个管理员账号单独对RabbitMQ进行管理。

```
[root@worker1 ~]# rabbitmqctl add_user admin admin  
Adding user "admin" ...  
Done. Don't forget to grant the user permissions to some virtual hosts! See  
'rabbitmqctl help set_permissions' to learn more.  
[root@worker1 ~]# rabbitmqctl set_permissions -p / admin "." "." ".*"  
Setting permissions for user "admin" in vhost "/" ...  
[root@worker1 ~]# rabbitmqctl set_user_tags admin administrator  
Setting tags for user "admin" to [administrator] ...
```

这样就可以用admin/admin用户登录Web控制台了。

登录控制台后上方就能看到RabbitMQ的主要功能。其中Overview是概述，主要展示RabbitMQ服务的一些整体运行情况。后面Connections、Channels、Exchanges和Queues就是RabbitMQ的核心功能。最后的Admin则是一些管理功能。

[Overview](#)
[Connections](#)
[Channels](#)
[Exchanges](#)
[Queues](#)
[Admin](#)

其中Admin主要是用来管理一些RabbitMQ的服务资源。例如刚才用命令行创建的admin用户，就可以在用户管理模块进行操作。

[Overview](#)
[Connections](#)
[Channels](#)
[Exchanges](#)
[Queues](#)
[Admin](#)

Cluster **rabbit@worker1**  
User **admin** [Log out](#)

## Users

▼ All users (2)

Pagination

Page 1 of 1 - Filter:  ☐ Regex ?

Displaying 2 items , page size up to: 100

Name	Tags	Can access virtual hosts	Has password
<b>admin</b>	administrator	/, /mirror	•
<b>guest</b>	administrator	/	•

?

► Add a user

Users

Virtual Hosts

Feature Flags

Policies

Limits

Cluster

[HTTP API](#)
[Server Docs](#)
[Tutorials](#)
[Community Support](#)
[Community Slack](#)
[Commercial Support](#)
[Plugins](#)
[GitHub](#)
[Changelog](#)

实际上RabbitMQ几乎所有的后台管理指令，都可以在管理页面上进行操作。

接下来可以尝试创建一个Virtual Hosts虚拟机。

[Overview](#)
[Connections](#)
[Channels](#)
[Exchanges](#)
[Queues](#)
[Admin](#)

Cluster **rabbit@worker1**  
User **admin** [Log out](#)

## Virtual Hosts

▼ All virtual hosts

Filter:  ☐ Regex ?

2 items, page size up to: 100

Overview			Messages			Network		Message rates	
Name	Users ?	State	Ready	Unacked	Total	From client	To client	publish	deliver / get
/	admin, guest	running	0	0	0				
/mirror	admin	running	1	0	1			0.00/s	0.00/s

▼ Add a new virtual host

Name:  \*

Description:

Tags:

Default Queue Type: Classic ▼

Add virtual host

Users

**Virtual Hosts**

Feature Flags

Policies

Limits

Cluster


这里就创建了一个名为/mirror的虚拟机，并配置了admin用户拥有访问的权限。在RabbitMQ中，不同虚拟机之间的资源是完全隔离的。不考虑资源分配的情况下，每个虚拟机就可以当成一个独立的RabbitMQ服务来使用。

管理页面的其他功能就不详细介绍了，后续随着深入使用再详细介绍

## 4、理解Exchange和Queue

Exchange和Queue是RabbitMQ中用来传递消息的核心组件。我们可以简单体验一下。

1、在Queues菜单，创建一个经典队列

 RabbitMQ™

RabbitMQ 3.11.10Erlang 25.2.2

Overview

Connections

Channels

Exchanges

Queues

Admin

## Queues

▼ All queues (0)

Pagination

Page 

▼

 of 0 - Filter: ☐ Regex 

?

... no queues ...

▼ Add a new queue

Virtual host:

/mirror ▼

Type:

Classic ▼

Name:

test1 \*

Durability:

Durable ▼

Auto delete: 

?

No ▼

Arguments:

= 

String ▼

Add 

Auto expire 

?

 | 

Message TTL 

?

 | 

Overflow behaviour 

?

 | 

Single active consumer 

?

 | 

Dead letter exchange 

?

 | 

Dead letter routing key 

?

 | 

Max length 

?

 | 

Max length bytes 

?

 | 

Maximum priority 

?

 | 

Lazy mode 

?

 | 

Version 

?

 | 

Master locator 

?

Add queue

创建完成后，选择这个test1队列，就可以在页面上直接发送消息以及消费消息了。

## Queue test1 in virtual host /mirror

- ▶ Overview
- ▶ Consumers (0)
- ▶ Bindings (1)
- ▶ Publish message 发消息
- ▶ Get messages 收消息
- ▶ Move messages
- ▶ Delete
- ▶ Purge
- ▶ Runtime Metrics (Advanced)

从这里可以看到，RabbitMQ中的消息都是通过Queue队列传递的，这个Queue其实就是一个典型的FIFO的队列数据结构。而Exchange交换机则是用来辅助进行消息分发的。Exchange与Queue之间会建立一种绑定关系，通过绑定关系，Exchange交换机里发送的消息就可以分发到不同的Queue上。

在整体使用过程中你会发现，对于队列，有Classic、Quorum、Stream三种类型，其中，Classic和Quorum两种类型，使用上几乎是没有什么区别的。但是Stream队列就无法直接消费消息了。并且每种不同类型的队列可选的参数也有很多不同。这种区别也会带到后面的使用过程中。

队列Queue即可以发消息，也可以收消息，那旁边的Exchange交换机是干什么的呢？其实他也是用来辅助发送消息的。

进入Exchanges菜单，可以看到针对每个虚拟机，RabbitMQ都预先创建了多个Exchange交换机。

Virtual host	Name	Type	Features	Message rate in	Message rate out	+/-
/	(AMQP default)	direct	D			
/	amq.direct	direct	D			
/	amq.fanout	fanout	D			
/	amq.headers	headers	D			
/	amq.match	headers	D			
/	amq.rabbitmq.trace	topic	D I			
/	amq.topic	topic	D			
/mirror	(AMQP default)	direct	D			
/mirror	amq.direct	direct	D			
/mirror	amq.fanout	fanout	D			
/mirror	amq.headers	headers	D			
/mirror	amq.match	headers	D			
/mirror	amq.rabbitmq.trace	topic	D I			
/mirror	amq.topic	topic	D			

这里我们选择amq.direct交换机，进入交换机详情页，选择Binding，并将test1队列绑定到这个交换机上。

## Exchange: amq.direct in virtual host /mirror

► Overview

▼ Bindings

This exchange



... no bindings ...

Add binding from this exchange

To queue ▼:

test1

\*

Routing key:

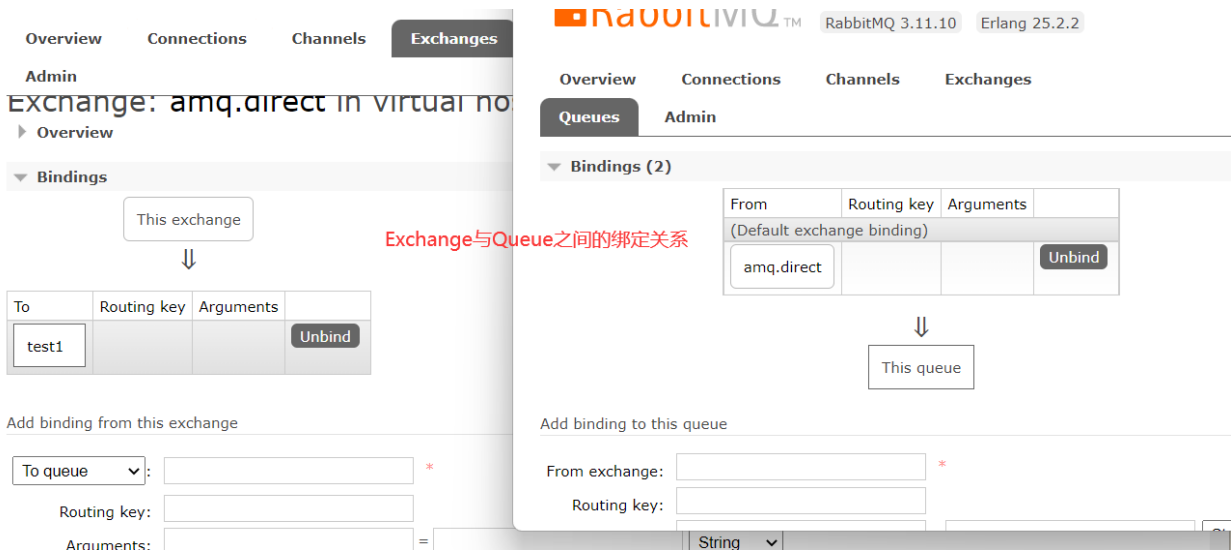
Arguments:

=

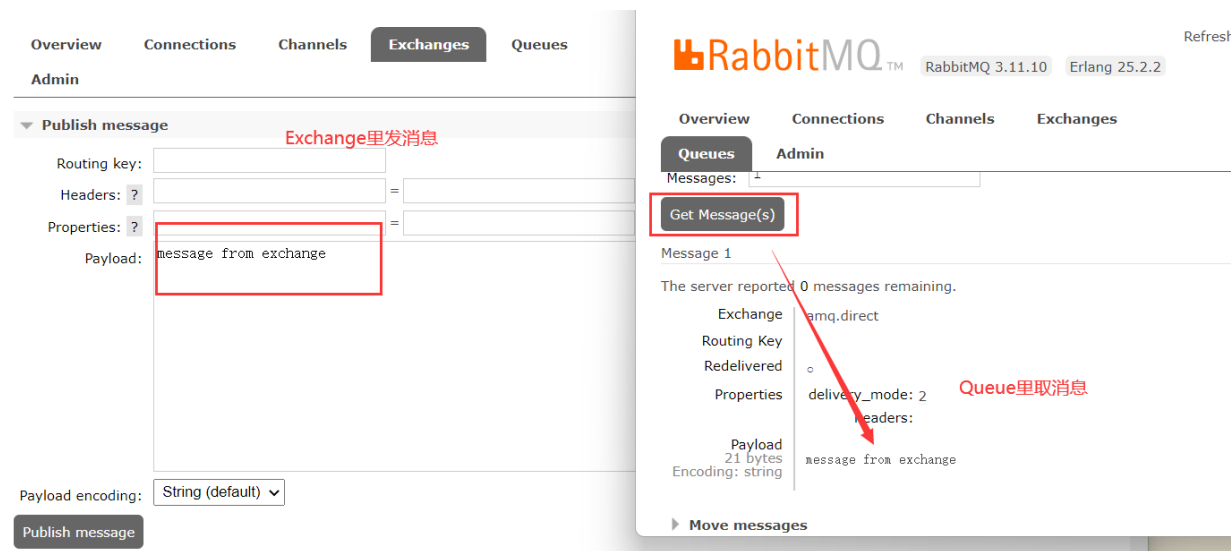
String ▼

Bind

绑定完成后，可以在Exchange详情页以及Queue详情页都看到绑定的结果。



接下来就可以在Exchange的详情页里发送消息。然后在test1这个queue里就能消费到这条消息。



Exchange交换机既然可以绑定一个队列，当然也可以绑定更多的队列。而Exchange的作用，就是将发送到Exchange的消息转发到绑定的队列上。在具体使用时，通常只有消息生产者需要与Exchange打交道。而消费者，则并不需要与Exchange打交道，只要从Queue中消费消息就可以了。

另外，Exchange并不只是简单的将消息全部转发给Queue，在实际使用中，Exchange与Queue之间可以建立不同类型的绑定关系，然后通过一些不同的策略，选择将消息转发到哪些Queue上。这时候，Message上几个没有用上的参数，像Routing Key, Headers, Properties这些参数就能派上用场了。

在这个过程中，我们都是通过页面操作完成的消息发送与接收。在实际应用时，其实就是通过RabbitMQ提供的客户端API来完成这些功能。但是整个执行的过程，其实跟页面操作是相同的。

**渔与鱼：**1、页面上的这些操作，建议你一定要自己动手，到处尝试尝试。不要觉得能收发消息了，对RabbitMQ就掌握得差不多了。并且，多带上脑子来体验，收集你在使用当中的困惑，这样在后续学习过程中你会更有目标。比如Stream类型的队列为什么不能在页面上获取消息？记住这个问题，后续在我给你分享队列类型时，你才有兴趣听得进去。



2、之前说过RabbitMQ的这些操作几乎都有后台命令行工具与之相匹配。而后台的很多命令行也提供了丰富的帮助文档。借助页面控制台了解后台命令行的操作，也是一个不错的方式。比如我们之前使用rabbitmqctl adduser指令添加了admin用户。那添加虚拟机、添加队列、添加交换机这些也可以类似去了解。你只需要使用rabbitmqctl -help 就能看到大量的帮助文档。

## 5、理解Connection和Channel

这两个概念实际上是跟客户端应用的对应关系。一个Connection可以理解为一个客户端应用。而一个应用可以创建多个Channel，用来与RabbitMQ进行交互。

我们可以来搭建一个客户端应用了解一下。

1、创建一个Maven项目，在pom.xml中引入RabbitMQ客户端的依赖：

```
<dependency>
  <groupId>com.rabbitmq</groupId>
  <artifactId>amqp-client</artifactId>
  <version>5.9.0</version>
</dependency>
```

2、然后就可以创建一个消费者实例，尝试从RabbitMQ上的test1这个队列上拉取消息。

```
public class FirstConsumer {
    private static final String HOST_NAME="192.168.65.112";
    private static final int HOST_PORT=5672;
    private static final String QUEUE_NAME="test2";
    public static final String USER_NAME="admin";
    public static final String PASSWORD="admin";
    public static final String VIRTUAL_HOST="/mirror";

    public static void main(String[] args) throws Exception{
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost(HOST_NAME);
        factory.setPort(HOST_PORT);
        factory.setUsername(USER_NAME);
        factory.setPassword(PASSWORD);
        factory.setVirtualHost(VIRTUAL_HOST);
        Connection connection = factory.newConnection();
        Channel channel = connection.createChannel();
        /**
         * 声明一个队列。几个参数依次为： 队列名，durable是否实例化；exclusive: 是否独占；
         * autoDelete: 是否自动删除；arguments: 参数
         * 这几个参数跟创建队列的页面是一致的。
         * 如果Broker上没有队列，那么就会自动创建队列。
         * 但是如果Broker上已经由了这个队列。那么队列的属性必须匹配，否则会报错。
         */
        channel.queueDeclare(QUEUE_NAME, true, false, false, null);
        //每个worker同时最多只处理一个消息
        channel.basicQos(1);
        //回调函数，处理接收到的消息
        Consumer myconsumer = new DefaultConsumer(channel) {
```

```

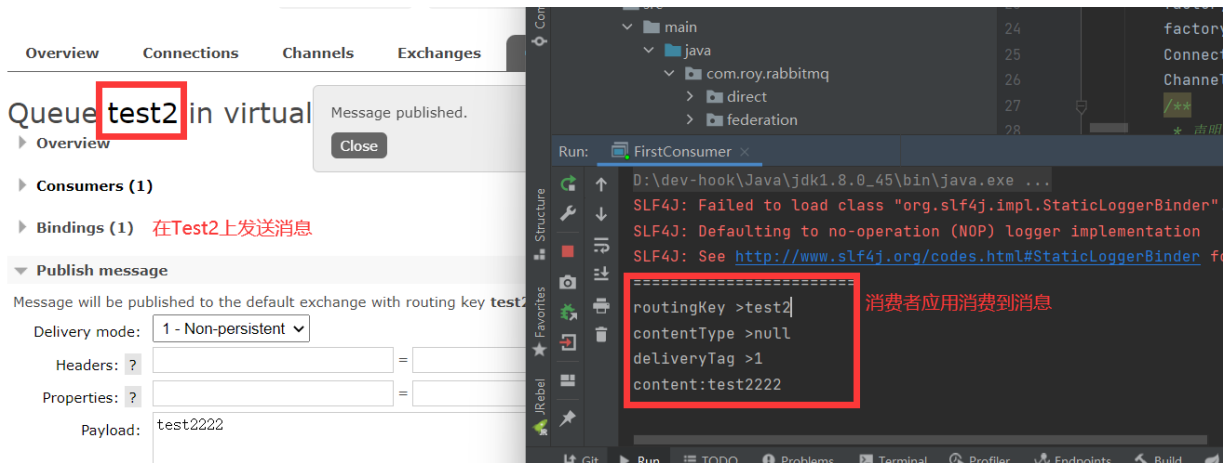
@Override
public void handleDelivery(String consumerTag, Envelope envelope,
                           AMQP.BasicProperties properties, byte[] body)
    throws IOException {
    System.out.println("=====");
    String routingKey = envelope.getRoutingKey();
    System.out.println("routingKey >"+routingKey);
    String contentType = properties.getContentType();
    System.out.println("contentType >"+contentType);
    long deliveryTag = envelope.getDeliveryTag();
    System.out.println("deliveryTag >"+deliveryTag);
    System.out.println("content:"+new String(body,"UTF-8"));
    // (process the message components here ...)
    channel.basicAck(deliveryTag, false);
}

};
//从test1队列接收消息
channel.basicConsume(QUEUE_NAME, myconsumer);
}
}

```

暂时不用过多纠结于实现细节，注意梳理整体实现流程。

执行这个应用程序后，就会在RabbitMQ上新创建一个test2的队列(如果你之前没有创建过的话)，并且启动一个消费者，处理test2队列上的消息。这时，我们可以从管理平台页面上往test2队列发送一条消息，这个消费者程序就会及时消费消息。



然后在管理平台的Connections和Channels里就能看到这个消费者程序与RabbitMQ建立的一个Connection连接与一个Channel通道。

图灵 百度 On Java架构师学习 【图灵VIP严选】 图灵VIP问答社

http://192.168.65.26:15672

# RabbitMQ™

RabbitMQ 3.11.10 Erlang 25.2.2

Overview Connections **Channels** Exchanges Queues Admin

Page 1 of 1 - Filter:  ☐ Regex ?

**通道**

Overview					Details		
Channel	Virtual host	User name	Mode ?	State	Unconfirmed	Prefetch ?	Un
192.168.65.26:11372 (1)	/mirror	admin		idle	0	1	

HTTP API Server Docs Tutorials Community Support Community Slack Comm

Overview				Details			Network		+/-
Virtual host	Name	User name	State	SSL / TLS	Protocol	Channels	From client	To client	
/mirror	192.168.65.26:11372 ?	admin	running		AMQP 0-9-1	1	0 B/s	0 B/s	

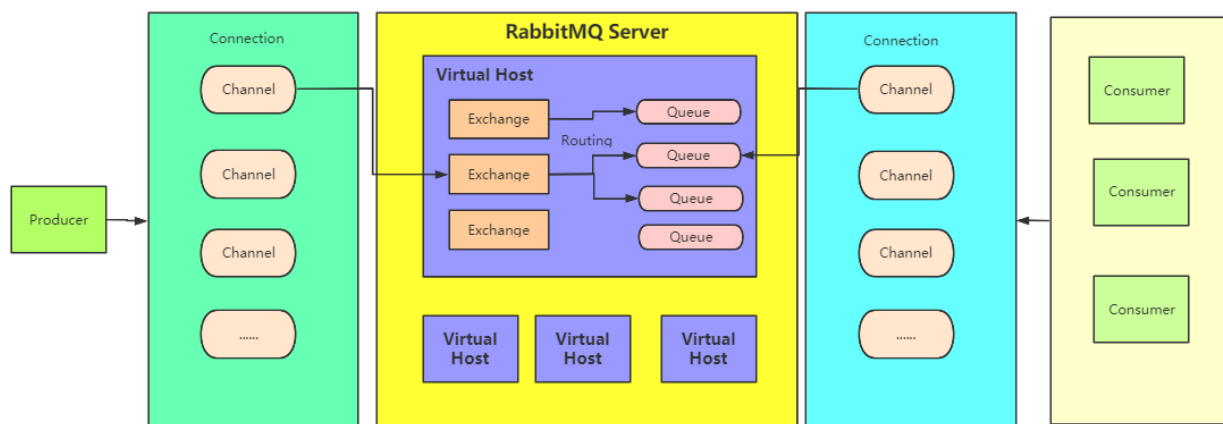
连接

这里可以看到Connection就是与客户端的一个连接。只要连接还通着，他的状态就是running。而Channel是RabbitMQ与客户端进行数据交互的一个通道，没有数据交互时，状态就是idle闲置。有数据交互时，就会变成running。在他们后面，都会展示出数据交互的状态。

另外，从这个简单示例中可以看到，Channel是从Connection中创建出来的，这也意味着，一个Connection中可以创建出多个Channel。从这些Connection和Channel中可以很方面的了解到RabbitMQ当前的服务运行状态。

## 6、RabbitMQ中的核心概念总结

通过这些操作，我们就可以了解到RabbitMQ的消息流转模型。



这里包含了很多RabbitMQ的重要概念：

### 1、服务主机Broker

一个搭建RabbitMQ Server的服务器称为Broker。这个并不是RabbitMQ特有的概念，但是却是几乎所有MQ产品通用的一个概念。未来如果需要搭建集群，就需要通过这些Broker来构建。

### 2、虚拟主机 virtual host

RabbitMQ出于服务器复用的想法，可以在一个RabbitMQ集群中划分出多个虚拟主机，每一个虚拟主机都有全套的基础服务组件，可以针对每个虚拟主机进行权限以及数据分配。不同虚拟主机之间是完全隔离的，如果不考虑资源分配的情况，一个虚拟主机就可以当成一个独立的RabbitMQ服务使用。

## 2、连接 Connection

客户端与RabbitMQ进行交互，首先就需要建立一个TCP连接，这个连接就是Connection。既然是通道，那就需要尽量注意在停止使用时要关闭，释放资源。

## 3、信道 Channel

一旦客户端与RabbitMQ建立了连接，就会分配一个AMQP信道 Channel。每个信道都会被分配一个唯一的ID。也可以理解是为客户端与RabbitMQ实际进行数据交互的通道，我们后续的大多数的数据操作都是在信道 Channel 这个层面展开的。

RabbitMQ为了减少性能开销，也会在一个Connection中建立多个Channel，这样便于客户端进行多线程连接，这些连接会复用同一个Connection的TCP通道，所以在实际业务中，对于Connection和Channel的分配也需要根据实际情况进行考量。

## 4、交换机 Exchange

这是RabbitMQ中进行数据路由的重要组件。消息发送到RabbitMQ中后，会首先进入一个交换机，然后由交换机负责将数据转发到不同的队列中。RabbitMQ中有多种不同类型的交换机来支持不同的路由策略。从Web管理界面就能看到，在每个虚拟主机中，RabbitMQ都会默认创建几个不同类型的交换机来。

Virtual host	Name	Type	Features	Message rate in	Message rate out	+/-
/	(AMQP default)	direct	D			
/	amq.direct	direct	D			
/	amq.fanout	fanout	D			
/	amq.headers	headers	D			
/	amq.match	headers	D			
/	amq.rabbitmq.trace	topic	D I			
/	amq.topic	topic	D			
/mirror	(AMQP default)	direct	D	0.00/s	0.00/s	
/mirror	amq.direct	direct	D	0.00/s	0.00/s	
/mirror	amq.fanout	fanout	D			
/mirror	amq.headers	headers	D			
/mirror	amq.match	headers	D			
/mirror	amq.rabbitmq.trace	topic	D I			
/mirror	amq.topic	topic	D			

交换机多用来与生产者打交道。生产者发送的消息通过Exchange交换机分配到各个不同的Queue队列上，而对于消息消费者来说，通常只需要关注自己感兴趣的队列就可以了。

## 5、队列 Queue

Queue是实际保存数据的最小单位。Queue不需要Exchange也可以独立工作，只不过通常在业务场景中，会增加Exchange实现更复杂的消息分配策略。Queue结构天生就具有FIFO的顺序，消息最终都会被分到不同的Queue当中，然后才被消费者进行消费处理。这也是最近RabbitMQ功能变动最大的地方。最为常用的是经典队列Classic。RabbitMQ 3.8.X版本添加了Quorum队列，3.9.X又添加了Stream队列。从官网的封面就能看到，现在RabbitMQ主推的是Quorum队列。

## 三、章节总结

这一章节最主要的，就是了解RabbitMQ的基础消息模型，这是RabbitMQ最重要最核心的部分。而后续就是针对这个大的模型，逐步深入，了解RabbitMQ的更多功能细节。

如果你之前已经使用过RabbitMQ，那么前面的操作你都可以忘记，但是最后这个大模型，你一定要先构建起来，并且可以尝试根据你的经验，看看可以往这个模型中添加一些什么内容。例如消息如何传递、销毁。Exchange与Queue之间如何进行消息路由等等。而如果你之前没有使用过RabbitMQ，那么前面的这些操作你最好自己动手都试试，并且在管理页面上到处多点点，多试试RabbitMQ的功能。不要说课上只创建了一个Queue，那你也照着创建一个Queue就完了。以可视化的方式构建起最后的消息模型，保证对于这个模型当中的各个重要概念的理解是准确的。

在这个章节中，我们只演示了Consumer端的示例代码，并没有演示Producer端的示例代码。但是，如果你熟悉了整个消息模型，也可以猜想得到Producer端的代码应该怎么写。无非也是这几个关键的步骤：

- 1、创建Connection
- 2、创建Channel
- 3、声明exchange(如果需要的话)
- 4、声明Queue
- 5、发送消息，可以发到Exchange，也可以发到Queue。

例如一个简单的发送者如下：

```
public class FirstProducer {

    private static final String HOST_NAME="192.168.65.112";
    private static final int HOST_PORT=5672;
    private static final String QUEUE_NAME="test2";
    public static final String USER_NAME="admin";
    public static final String PASSWORD="admin";
    public static final String VIRTUAL_HOST="/mirror";

    public static void main(String[] args) throws Exception{
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost(HOST_NAME);
        factory.setPort(HOST_PORT);
        factory.setUsername(USER_NAME);
        factory.setPassword(PASSWORD);
        factory.setVirtualHost(VIRTUAL_HOST);
        Connection connection = factory.newConnection();
        Channel channel = connection.createChannel();
        /**
         * 声明一个队列。几个参数依次为： 队列名，durable是否实例化；exclusive：是否独占；
         * autoDelete：是否自动删除；arguments：参数
         * 这几个参数跟创建队列的页面是一致的。
         */
    }
}
```

```
    * 如果Broker上没有队列，那么就会自动创建队列。  
    * 但是如果Broker上已经由了这个队列。那么队列的属性必须匹配，否则会报错。  
    */  
    channel.queueDeclare(QueueName, true, false, false, null);  
    String message = "message";  
    channel.basicPublish("", QueueName,  
        MessageProperties.PERSISTENT_TEXT_PLAIN, message.getBytes());  
  
    channel.close();  
    connection.close();  
}  
}
```

你看，整个流程跟消费者是不是差不多的？除了生产者发送完消息后需要主动关闭下连接，而消费者因为要持续消费消息所以不需要主动关闭连接，其他流程几乎完全一样的。

有道云笔记链接地址：<https://note.youdao.com/s/9EDfKHZ9>