# 一、RabbitMQ 基本概念

## 1.1 ConnectionFactory、Connection、Channel

- ConnectionFactory、Connection、Channel 都是 RabbitMQ 对外提供的 API中 最基本的对象。
  - Connection 是 **RabbitMQ 的 socket 链接**，它封装了 socket 协议相关部分逻辑。
  - ConnectionFactory 为 Connection 的创建工厂。
  - Channel 是我们与 RabbitMQ 打交道的最重要的一个接口，我们大部分的业务操作是在 Channel 这个接口中完成的，包括定义 Queue、定义 Exchange、绑定 Queue 与 Exchange、发布消息等。
- **为什么使用 Channel，而不是直接使用 TCP 连接?**
  - 对于 OS 来说，建立和关闭 TCP 连接是有代价的，频繁的建立关闭 TCP 连接对于系统的性能有很大的影响，而且 TCP 的连接数也有限制，这也限制了系统处理高并发的能力。
  - 但是，**在 TCP 连接中建立 Channel 是没有上述代价的**。对于 Producer 或者 Consumer 来说，可以并发的使用多个 Channel 进行 Publish 或者 Receive。
  - 有实验表明，1s 的数据可以 Publish 10K 的数据包。当然对于不同的硬件环境，不同的数据包大小这个数据肯定不一样，而对于普通的 Consumer 或者 Producer 来说，这已经足够了。如果不够用，你考虑的应该是如何细化 split 你的设计。

## 1.2 Queue

- Queue（队列）是 RabbitMQ 的内部对象，用于存储消息。
- **RabbitMQ 中的消息都只能存储在 Queue 中**，生产者生产消息并最终投递到 Queue 中，消费者可以从 Queue 中获取消息并消费。
- 多个消费者可以订阅同一个 Queue，这时 Queue 中的消息会被**平均分摊**给多个消费者进行处理，而不是每个消费者都收到所有的消息并处理。
- **声明 MessageQueue**

  - 在 RabbitMQ 中，无论是生产者发送消息还是消费者接受消息，都首先需要声明一个 MessageQueue。这就存在一个问题，是生产者声明还是消费者声明呢？要解决这个问题，首先需要明确：
    - 消费者无法订阅或者获取不存在的 MessageQueue 中的信息。
    - **消息被 Exchange 接受以后，如果没有匹配的 Queue，则会被丢弃。**
  - 如果是消费者去声明 Queue，就有可能会出现在声明 Queue 之前，生产者已发送的消息被丢弃的隐患。
  - 如果一个消费者在一个信道(channel)中正在监听某一个队列的消息，RabbitMQ 是不允许该消费者在同一个 channel 去声明其他队列的。
  - RabbitMQ 中，可以通过 queue.declare 命令声明一个队列，可以设置该队列以下属性:

    1. **Exclusive：排他队列**，如果一个队列被声明为排他队列，该队列仅对首次声明它的连接可见，并在连接断开时自动删除。需要注意三点：
       1. **排他队列是基于连接可见的**，同一连接的不同信道是可以同时访问同一个连接创建的排他队列的。
       2. “首次”，如果一个连接已经声明了一个排他队列，其他连接是不允许建立同名的排他队列的，这个与普通队列不同。
       3. 即使该队列是持久化的，一旦连接关闭或者客户端退出，该排他队列都会被自动删除的。这种队列适用于只限于一个客户端发送读取消息的应用场景。

> 2. **Auto-delete: 自动删除**，如果该队列没有任何订阅的消费者的话，该队列会被自动删除。这种队列适用于临时队列。
> 3. **Durable: 持久化。**
> 4. 其他选项，例如如果用户仅仅想**查询某一个队列是否已存在**，如果不存在，不想建立该队列，仍然可以调用 queue.declare，只不过需要将参数 passive 设为 true，传给 queue.declare，如果该队列已存在，则会返回 true；如果不存在，则会返回 Error，但是不会创建新的队列。
>
> ◦

## 1.3 Message acknowledgment（ack）

- 在实际应用中，可能会发生消费者收到 Queue 中的消息，但没有处理完成就宕机（或出现其他意外）的情况，这种情况下就可能会导致消息丢失。
- 为了避免这种情况发生，我们可以要求消费者在消费完消息后发送一个回执给 RabbitMQ，RabbitMQ 收到消息回执（Message acknowledgment）后才将该消息从 Queue 中移除；
- 如果 RabbitMQ 没有收到回执并检测到消费者的 RabbitMQ 连接断开，则 RabbitMQ 会将该消息发送给其他消费者（如果存在多个消费者）进行处理。这里不存在 timeout 概念，**一个消费者处理消息时间再长也不会导致该消息被发送给其他消费者，除非它的RabbitMQ连接断开。**
- 这里会产生另外一个问题，如果我们的开发人员在处理完业务逻辑后，**忘记发送回执给 RabbitMQ**，这将会导致严重的 bug——Queue 中堆积的消息会越来越多；消费者重启后会重复消费这些消息并重复执行业务逻辑…

## 1.4 Message durability（持久化）

- 如果我们希望即使在 RabbitMQ 服务重启的情况下，也不会丢失消息，我们可以将 Queue 与 Message 都设置为可持久化的（durable），这样可以保证绝大部分情况下我们的 RabbitMQ 消息不会丢失。
- 但依然解决不了小概率丢失事件的发生（比如 RabbitMQ 服务器已经接收到生产者的消息，但还没来得及持久化该消息时 RabbitMQ 服务器就断电了），如果我们需要对这种小概率事件也要管理起来，那么我们要用到事务。

## 1.5 Prefetch count（消息处理时间不同）

- 如果有多个消费者同时订阅同一个 Queue 中的消息，Queue 中的消息会被平摊给多个消费者。这时如果每个消息的处理时间不同，就有可能会导致某些消费者一直在忙，而另外一些消费者很快就处理完手头工作并一直空闲的情况。
- 我们可以通过设置 prefetchCount 来限制 Queue 每次发送给每个消费者的消息数，比如我们设置 prefetchCount = 1，则 Queue 每次给每个消费者发送一条消息；消费者处理完这条消息后 Queue 会再给该消费者发送一条消息。

## 1.6 routing key

- 生产者在将消息发送给 Exchange 的时候，一般会指定一个routing key，来指定这个消息的路由规则，而这个 routing key 需要与 Exchange Type 及 binding key 联合使用才能最终生效。
- 在 Exchange Type 与 binding key 固定的情况下（在正常使用时一般这些内容都是固定配置好的），我们的**生产者就可以在发送消息给 Exchange 时，通过指定 routing key 来决定消息流向哪里。**
- RabbitMQ 为 routing key 设定的长度限制为 255 bytes。

## 1.7 Binding

- RabbitMQ 中通过 Binding 将 Exchange 与 Queue 关联起来，这样 RabbitMQ 就知道如何正确地将消息路由到指定的 Queue 了。

## 1.8 Binding key

- 在绑定（Binding）Exchange 与 Queue 的同时，一般会指定一个 binding key；消费者将消息发送给 Exchange 时，一般会指定一个 routing key；当 binding key 与 routing key 相匹配时，消息将会被路由到对应的 Queue 中。
- 在绑定多个 Queue 到同一个 Exchange 的时候，这些 Binding 允许使用相同的 binding key。
- binding key 并不是在所有情况下都生效，它依赖于 Exchange Type，比如 **fanout 类型的 Exchange 就会无视 binding key，而是将消息路由到所有绑定到该Exchange 的 Queue。**

## 1.9 Exchange 类型

RabbitMQ 常用的 Exchange 类型有 fanout、direct、topic、headers 这四种（AMQP规范里还提到两种Exchange Type，分别为 system 与自定义）

- fanoutExchange
    - fanout 类型的 Exchange 路由规则非常简单，它会把所有发送到该 Exchange 的消息路由到所有与它绑定的 Queue 中。
    - 需要创建临时队列，才能接受所有消息。====》相当于将消息同时发送到多个临时队列上，否则是在同一个队列上消费。
    - 先声明临时队列，再启动生产者，消费者才能消费到消息。
- directExchange
    - direct 类型的 Exchange 路由规则也很简单，它会把消息路由到那些 binding key 与 routing key 完全匹配的 Queue 中。
    - **DirectExchange 实现了轮询的方式对消息进行消费，而且不存在重复消费。**
- topicExchange
    - direct 类型的 Exchange 路由规则是完全匹配 binding key 与 routing key，但这种严格的匹配方式在很多情况下不能满足实际业务需求。
    - topic 类型的 Exchange 在匹配规则上进行了扩展，它与 direct 类型的 Exchage相似，也是将消息路由到 binding key 与 routing key 相匹配的Queue中，但这里的匹配规则有些不同，它约定：
        - routing key 为一个句点号“. ”分隔的字符串（我们将被句点号“. ”分隔开的每一段独立的字符串称为一个单词），如“stock.usd.nyse”、“nyse.vmw”、“quick.orange.rabbit”
        - binding key与routing key一样也是句点号“. ”分隔的字符串
        - binding key中可以存在两种特殊字符“*”*与*“#”，用于做模糊匹配，其中“*”*用于匹配一个单词，“#”用于匹配多个单词（可以是零个）
- headers
    - headers 类型的 Exchange 不依赖于 routing key 与 binding key 的匹配规则来路由消息，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 Exchange 时，RabbitMQ 会取到该消息的 headers（也是一个键值对的形式），对比其中的键值对是否完全匹配 Queue 与 Exchange 绑定时指定的键值对；如果完全匹配则消息会路由到该 Queue，否则不会路由到该 Queue。

## 1.10 Vhost

- vhost 本质上是一个 mini 版的 RabbitMQ 服务器，拥有自己的 connection、queue（队列）、binding（绑定）、exchange（交换器）、user permission（权限控制）、policies（策略）。
- vhost 通过在各个实例间**提供逻辑上分离**，允许你为不同应用程序安全保密地运行数据。

- vhost 是 AMQP 概念的基础，必须在连接时进行指定，RabbitMQ 包含了默认 vhost：“/”；
- 当在 RabbitMQ 中创建一个用户时，用户通常会被指派给至少一个 vhost，并且只能访问被指派 vhost 内的队列、交换器和绑定，**vhost 之间是绝对隔离的。**

## 1.11 RPC

- MQ 本身是基于异步的消息处理，所有的生产者（P）将消息发送到 RabbitMQ 后不会知道消费者（C）处理成功或者失败（甚至连有没有消费者来处理这条消息都不知道）。
- 但实际的应用场景中，我们很可能需要一些同步处理，需要同步等待服务端将我的消息处理完成后再进行下一步处理。这相当于 RPC（Remote Procedure Call，远程过程调用）。在 RabbitMQ 中也支持 RPC。
- **RabbitMQ 中实现 RPC 的机制是：**
    - 客户端发送请求（消息）时，在消息的属性（MessageProperties，在AMQP协议中定义了 14 种 properties，这些属性会随着消息一起发送）中设置两个值 **replyTo**（一个Queue名称，用于告诉服务器处理完成后将通知我的消息发送到这个 Queue 中）和 **correlationId**（此次请求的标识号，服务器处理完成后需要将此属性返还，客户端将根据这个 id 了解哪条请求被成功执行了或执行失败）
    - 服务器端收到消息并处理
    - 服务器端处理完消息后，将生成一条应答消息到 replyTo 指定的 Queue，同时带上 correlationId 属性
    - 客户端之前已订阅 replyTo 指定的 Queue，从中收到服务器的应答消息后，根据其中的 correlationId 属性分析哪条请求被执行了，根据执行结果进行后续业务处理

## 1.12 RabbitMQ Broker

- 也叫 broker server，是一种传输服务。
- 他的角色就是维护一条从 Producer 到 Consumer 的路线，保证数据能够按照指定的方式进行传输。这个保证不是 100% 的保证，但是对于普通的应用来说这已经足够了。
- 对于商业系统来说，可以再做一层数据一致性的 guard，就可以彻底保证系统的一致性了。

## 1.13 Producer

- Producer，数据的发送方。Producer 创建并发布消息到 broker server 中。
- 一个 Message 有两个部分：payload（有效载荷）和 label（标签）。payload 是传输的数据。label 是 exchange 的名字或者说是一个 tag，它描述了 payload，而且 RabbitMQ 也是通过这个 label 来决定把这个 Message 发给哪个 Consumer。AMQP 仅仅描述了 label，而 RabbitMQ 决定了如何使用这个 label 的规则。
- **生产者发送消息**

> - 在 AMQP 模型中，**Exchange 是接受生产者消息并将消息路由到消息队列的关键组件。**ExchangeType 和 Binding 决定了消息的路由规则。所以生产者想要发送消息，首先必须要声明一个Exchange 和该 Exchange 对应的 Binding。可以通过 ExchangeDeclare 和 BindingDeclare 完成。
> - 在 RabbitMQ 中，声明一个 Exchange 需要三个参数：ExchangeName，ExchangeType 和 Durable。
>     - ExchangeName 是该 Exchange 的名字，该属性在创建 Binding 和生产者通过 publish 推送消息时需要指定。
>     - ExchangeType，指 Exchange 的类型，在 RabbitMQ 中，主要有三种类型的 Exchange：direct ， fanout 和 topic，不同的 Exchange 会表现出不同路由行为。

    - Durable 是该 Exchange 的持久化属性，在需要做消息持久化时指定。声明一个 Binding 需要提供一个 QueueName，ExchangeName 和 BindingKey（RoutingKey）。

## 1.14 Consumer

- Consumer，数据的接收方。Consumers 绑定一个 broker server 并订阅一个队列
- 在经过 broker server 的 Message 中，只有 payload，label 已经被删掉了。**对于 Consumer 来说，它是不知道谁发送的这个信息的。**协议本身不支持。如果 Producer 发送的 payload 包含了 Producer 的信息就另当别论了。
- **消费者订阅消息**
    - 在 RabbitMQ 中消费者有 2 种方式获取队列中的消息:
        1. 一种是通过 basic.consume 命令，订阅某一个队列中的消息，channel 会自动在处理完上一条消息之后，接收下一条消息。（同一个 channel 消息处理是串行的）。除非关闭 channel 或者取消订阅，否则客户端将会一直接收队列的消息。
        2. 另外一种方式是通过 basic.get 命令主动获取队列中的消息，但是绝对不可以通过循环调用 basic.get 来代替 basic.consume，这是因为 basic.get RabbitMQ 在实际执行的时候，是首先 consume 某一个队列，然后检索第一条消息，然后再取消订阅。如果是高吞吐率的消费者，最好还是建议使用 basic.consume。
    - 如果有多个消费者同时订阅同一个队列的话，RabbitMQ 是**采用循环（轮询）的方式分发消息**的，每一条消息只能被一个订阅者接收。

## 1.15 消息持久化（Exchange、Queue、Message）

- RabbitMQ 默认是**不持久 Queue、Exchange、Binding 以及队列中的消息**的，这意味着一旦消息服务器重启，所有已声明的队列、Exchange、Binding 以及队列中的消息都会丢失。
- 通过设置 Exchange 和 MessageQueue 的 durable 属性为 true，可以使得队列和 Exchange 持久化，但是这还不能使得队列中的消息持久化，这需要生产者在发送消息的时候，将 delivery mode 设置为2，只有这 3 个全部设置完成后，才能保证服务器重启不会对现有的队列造成影响。
- 这里需要注意的是，只有 durable 为 true 的 Exchange 和 durable 为 true 的 Queues 才能绑定，否则在绑定时，RabbitMQ 都会抛错的。持久化会对 RabbitMQ 的性能造成比较大的影响，可能会下降 10 倍不止。

## 1.16 事务

- 对事务的支持是 AMQP 协议的一个重要特性。假设当生产者将一个持久化消息发送给服务器时，因为 consume 命令本身没有任何 Response 返回，所以即使服务器崩溃，没有持久化该消息，生产者也无法获知该消息已经丢失。
- 如果此时使用事务，即通过 txSelect() 开启一个事务，然后发送消息给服务器，然后通过 txCommit() 提交该事务，即可以保证，如果 txCommit() 提交了，则该消息一定会持久化，如果 txCommit() 还未提交即服务器崩溃，则该消息不会服务器就收。RabbitMQ 也提供了 txRollback() 命令用于回滚某一个事务。

## 1.17 Publisher Confirm机制

- 使用事务固然可以保证只有提交的事务，才会被服务器执行。但是这样同时也将客户端与消息服务器同步起来，这**背离了消息队列解耦的本质。**
- RabbitMQ 提供了一个更加**轻量级的机制**来保证生产者可以感知服务器消息是否已被路由到正确的队列中——**Confirm。**如果设置 channel 为 confirm 状态，则通过该 channel 发送的消息都会被分配一个唯一的 ID，然后一旦该消息被正确的路由到匹配的队列中后，服务器会返回给生产者一

个 Confirm，该 Confirm 包含该消息的 ID，这样生产者就会知道该消息已被正确分发。**对于持久化消息，只有该消息被持久化后，才会返回 Confirm。**

- **Confirm 机制的最大优点在于异步**，生产者在发送消息以后，即可继续执行其他任务。而服务器返回 Confirm 后，会触发生产者的回调函数，生产者在回调函数中处理 Confirm 信息。如果消息服务器发生异常，导致该消息丢失，会返回给生产者一个 nack，表示消息已经丢失，这样生产者就可以通过重发消息，保证消息不丢失。Confirm 机制在性能上要比事务优越很多。
- 但是 Confirm 机制，无法进行回滚，就是一旦服务器崩溃，生产者无法得到 Confirm 信息，生产者其实本身也不知道该消息是否已经被持久化，只有继续重发来保证消息不丢失，但是如果原先已经持久化的消息，并不会被回滚，这样队列中就会存在两条相同的消息，**系统需要支持去重。**

## 1.18 Shovel

- Shovel 能够可靠、持续地从一个 Broker 中的队列（作为源端，即 source）拉取数据并转发至另一个 Broker 中的交换器（作为目的端，即 destination）。
  作为源端的队列和作为目的端的交换器可以同时位于同一个 Broker 上，也可以位于不同的 Broker 上。
- **Shovel 的使用**

  > - Shovel 插件默认也在 RabbitMQ 的发布包中，执行 `rabbitmq-plugins enable rabbitmq_shovel` 命令可以开启 Shovel 功能。
  > - 开启 rabbitmq shovel management 插件之后， 在 RabbitMQ 的管理界面中" Admin"的右侧会多出"Shovel Status" 和" Shovel Management " 两个 Tab 页。 `rabbitmq-plugins enable rabbitmq_shovel_management`
  > - Shovel 既可以部署在源端，也可以部署在目的端。有两种方式可以部署 Shovel：
  >   - 静态方式：在 `rabbitmq.config` 配置文件中设置
  >   - 动态方式：通过 Runtime Parameter 设置

## 1.19 Policy

- Policies 的作用
  - 通常来说，我们会在创建一个 Queue 时指定了队列的各项属性和参数，例如 message-ttl、x-dead-letter-exchange、x-dead-letter-route-key、x-max-length 等，**一旦 RabbitMQ 创建Queue成功，则不允许再修改 Queue 的任何属性或参数，除非删除重建。** 我们**在实际使用中，随着业务的变化，经常会出现需要调整 message-ttl、x-max-length 等参数来改变 Queue 行为的场景**，这个时候就需要使用到 RabbitMQ 的 Policy 机制。
- Operator Policy 和 User Policy 的区别：
  1. Operator Policy 是给服务提供商或公司基础设施部门用来设置某些需要强制执行的通用规则
  2. User Policy 是给业务应用用来设置的规则
- Policies 之间的优先级：在设置 Policy 时需要注意，因为 Policy 是根据 parttern 匹配队列的，因此可能会出现多个 Policy 都匹配到某一个队列的情况，此时会依据以下规则进行排序选出实际生效的 Policy：
  1. 首先根据 priority 排序，值越大的优先级越高；
  2. 相同 priority 则根据创建时间排序，越晚创建的优先级越高。

## 1.20 rabbitmq 脑裂问题

- **实质上是个网络分区问题**， 确切来说是网络不稳定导致的问题。rabbitmq集群的网络分区容错性不好，在网络比较差的情况下容易出错，最明显的就是脑裂问题了。
- 所谓的脑裂问题，就是在多机集群中节点与节点之间失联，**都认为对方出现故障，而自身裂变为独立的个体，各自为政，那么就出现了抢夺对方的资源**，争抢启动，至此就发生了事故。

- 如何恢复
  - 人力干扰
    1. 部分重启。选择受信分区即网络环境好的分区，重启其他分区，让他们重新加入受信分区中，之后再重启受信分区，消除警告⚠
    2. 全部重启。停掉整个集群，然后启动，不过要**保证第一个启动的是受信分区**。不然还没等所有节点加入，自己就挂了。
  - 自动处理（rabbitmq 提供了3种模式）
    1. ignore 默认配置，即分区不做任何处理。要使用这种，就要保证网络高可用，例如，节点都在一个机架上，用的同一个交换机，这个交换机连上一个WAN，保证网络稳定。
    2. pause_minority。优先暂停'少数派'。就是节点判断自己在不在'少数派'（少于或者等于集群中一半的节点数），在就自动关闭，保证稳定区的大部分节点可以继续运行。
    3. autoheal， 关闭客户端连接数最多的节点。 这里比较有意思，rabbitmq 会自动挑选一个'获胜'分区，即连接数最小的，重启其他分区。（如果平手，就选节点多的，如果节点也一致，那就以未知方式挑选'获胜者'）。这个更关心服务的连续性而不是数据的完整性。

## 1.21 RabbitMQ 端口解析

- **4369 (epmd), 25672 (Erlang distribution)**

  Epmd 是 Erlang Port Mapper Daemon 的缩写，在 Erlang 集群中相当于 dns 的作用，绑定在4369端口上。

  **4369 -- erlang 发现口　　25672 -- server 间内部通信口**

- **15672 (if management plugin is enabled)**

  通过 `http://serverip:15672` 访问 RabbitMQ 的 Web 管理界面，默认用户名密码都是 guest。

  15672 -- 管理界面 ui 端口

- **5672, 5671 (AMQP 0-9-1 without and with TLS)**

  基于 AMQP 协议的客户端与消息中间件之间可以传递消息，并不受客户端/中间件不同产品、不同的开发语言等条件的限制。

  5672 -- 客户端通信口

## 1.22 死信队列

- 什么是死信队列
  - "死信"是 RabbitMQ 中的一种消息机制，当你在消费消息时，如果队列里的消息出现以下情况：
    1. 消息被否定确认，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 TTL 时间。
    3. 消息队列的消息数量已经超过最大队列长度。
  - 那么该消息将成为"死信"。"死信"消息会被 RabbitMQ 进行特殊处理，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。
- 如何配置死信队列
  - 如何配置死信队列呢？大概可以分为以下步骤：
    1. 配置业务队列，绑定到业务交换机上
    2. 为业务队列配置死信交换机和 RoutingKey
    3. 为死信交换机配置死信队列

    - 为每个需要使用死信的业务队列配置一个死信交换机，这里同一个项目的死信交换机可以共用一个，然后为每个业务队列分配一个单独的路由 key。
    - **死信队列并不是什么特殊的队列，只不过是绑定在死信交换机上的队列。死信交换机也不是什么特殊的交换机，只不过是用来接受死信的交换机，所以可以为任何类型【Direct、Fanout、Topic】。**一般来说，会为每个业务队列分配一个独有的路由key，并对应的配置一个死信队列进行监听，也就是说，一般会为每个重要的业务队列配置一个死信队列。
- 死信消息的变化
    - “死信”被丢到死信队列中后，会发生什么变化呢？

        > - 如果队列配置了参数 `x-dead-letter-routing-key` 的话，“死信”的路由key将会被替换成该参数对应的值。如果没有设置，则保留该消息原有的路由key。
        > - 另外，由于被抛到了死信交换机，所以消息的 Exchange Name 也会被替换为死信交换机的名称。
- 死信队列应用场景

    > - 一般用在较为重要的业务队列中，确保未被正确消费的消息不被丢弃，一般发生消费异常可能原因主要有由于消息信息本身存在错误导致处理异常，处理过程中参数校验异常，或者因网络波动导致的查询异常等等，当发生异常时，当然不能每次通过日志来获取原消息，然后让运维帮忙重新投递消息（没错，以前就是这么干的= =）。
    > - 通过配置死信队列，可以让未正确处理的消息暂存到另一个队列中，待后续排查清楚问题后，编写相应的处理代码来处理死信消息，这样比手工恢复数据要好太多了。
- 总结
    - 死信队列是绑定在死信交换机上的普通队列，而死信交换机也只是一个普通的交换机，不过是用来专门处理死信的交换机。
    - **死信消息的生命周期**：
        1. 业务消息被投入业务队列
        2. 消费者消费业务队列的消息，由于处理过程中发生异常，于是进行了 nack 或者 reject 操作
        3. 被 nack 或 reject 的消息由 RabbitMQ 投递到死信交换机中
        4. 死信交换机将消息投入相应的死信队列
        5. 死信队列的消费者消费死信消息
    - 死信消息是 RabbitMQ 为我们做的一层保证，其实我们也可以不使用死信队列，而是在消息消费异常时，将消息主动投递到另一个交换机中，当你明白了这些之后，这些 Exchange 和 Queue 想怎样配合就能怎么配合。比如从死信队列拉取消息，然后发送邮件、短信、钉钉通知来通知开发人员关注。或者将消息重新投递到一个队列然后设置过期时间，来进行延时消费。

## 1.23 RabbitMQ 的 TTL

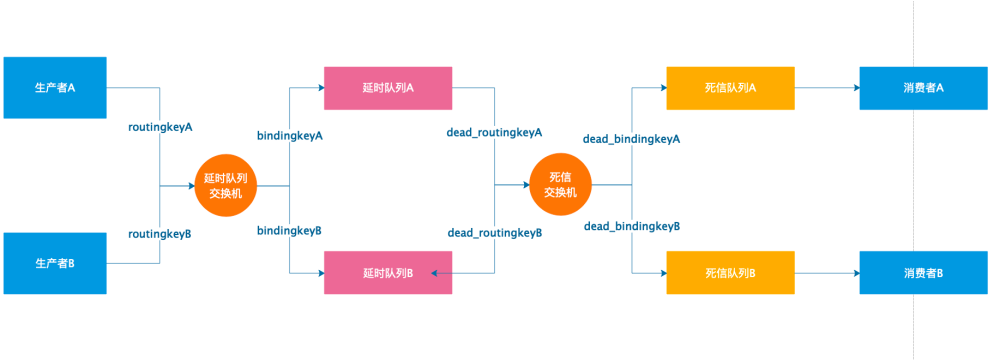
- `TTL` 是 RabbitMQ 中一个消息或者队列的属性，表明**一条消息或者该队列中的所有消息的最大存活时间**，单位是毫秒。
- 换句话说，如果一条消息设置了 TTL 属性或者进入了设置 TTL 属性的队列，那么这条消息如果在 TTL 设置的时间内没有被消费，则会成为“死信”。**如果同时配置了队列的 TTL 和消息的 TTL，那么较小的那个值将会被使用。**
- 设置 TTL 值的两种方式：
    1. 在创建**队列**的时候设置队列的 “x-message-ttl” 属性；
    2. 针对每条**消息**设置 TTL
    3. 区别：**如果设置了队列的TTL属性，那么一旦消息过期，就会被队列丢弃，而第二种方式，消息即使过期，也不一定会被马上丢弃，因为消息是否过期是在即将投递到消费者之前判定**

**的，如果当前队列有严重的消息积压情况，则已过期的消息也许还能存活较长时间。**

- 如果**不设置** TTL，表示消息永远不会过期，如果将 TTL **设置为 0**，则表示除非此时可以直接投递该消息到消费者，否则该消息将会被丢弃。

## 1.24 延时队列

- 什么是延时队列
  - `延时队列`，最重要的特性就体现在它的**延时**属性上，跟普通的队列不一样的是，**普通队列中的元素总是等着希望被早点取出处理，而延时队列中的元素则是希望被在指定时间得到取出和处理**，所以延时队列中的元素是都是带时间属性的，通常来说是需要被处理的消息或者任务。
  - 简单来说，延时队列就是用来存放需要在指定时间被处理的元素的队列。
- 使用场景
  - 需要在某个事件发生之后或者之前的指定时间点完成某一项任务的场景，如：
    1. 发生订单生成事件，在十分钟之后检查该订单支付状态，然后将未支付的订单进行关闭；
    2. 发生店铺创建事件，十天后检查该店铺上新商品数，然后通知上新数为 0 的商户；
    3. 发生账单生成事件，检查账单支付状态，然后自动结算未支付的账单；
    4. 发生新用户注册事件，三天后检查新注册用户的活动数据，然后通知没有任何活动记录的用户；
- 利用 RabbitMQ 实现延时队列
  - `延时队列`，就是要消息延迟一段时间后被处理，TTL 则刚好能让消息在延迟多久之后成为死信，另一方面，成为死信的消息都会被投递到死信队列里，这样只需要**消费者一直消费死信队列里的消息**就可以了，因为里面的消息都是希望被立即处理的消息。




- 问题（优化）：
  1. 如果**在队列上设置 TTL**，在使用的时候，每增加一个新的时间需求，就要新增一个队列。
  2. **将 TTL 设置在消息属性。**如果使用在消息属性上设置 TTL 的方式，消息可能并不会按时“死亡“，因为 RabbitMQ 只会检查第一个消息是否过期，如果过期则丢到死信队列，索引如果第一个消息的延时时长很长，而第二个消息的延时时长很短，则第二个消息并不会优先得到执行。
- 利用 RabbitMQ 插件实现延迟队列
  - 解决上述问题：安装一个插件即可：https://www.rabbitmq.com/community-plugins.html ，下载 rabbitmq_delayed_message_exchange 插件，然后解压放置到RabbitMQ的插件目录。
  - 接下来，进入 RabbitMQ 的安装目录下的 sbin 目录，执行命令让该插件生效，然后重启 RabbitMQ。
- 总结
  - **延时队列在需要延时处理的场景下非常有用，使用 RabbitMQ 来实现延时队列可以很好的利用 RabbitMQ 的特性，如：消息可靠发送、消息可靠投递、死信队列来保障消息至少被消费一次以及未被正确处理的消息不会被丢弃。**

    - 另外，通过 RabbitMQ 集群的特性，可以很好的解决单点故障问题，不会因为单个节点挂掉导致延时队列不可用或者消息丢失。

# 二、AMQP 协议

- AMQP把客户端和代理服务器之间的通信数据拆分成一种叫作帧（frame）的块结构

## 2.1 AMQP作为一种RPC传输机制

- 一个AMQP连接可以有多个信道，允许客户端和服务器之间进行多次会话。从技术上讲，这被称为**多路复用（multiplexing）**，对于执行多个任务的多线程或异步应用程序来说，它非常有用。
- 在创建客户端应用程序时，不要使用过多的信道使事情变得复杂。

## 2.2 AMQP RPC帧结构

- AMQP 使用类和方法在客户端和服务器之间创建公共语言，这些类和方法被称为 **AMQP 命令（AMQP commands）**。

### 2.2.1 AMQP帧组件

- 当使用命令与 RabbitMQ 进行交互时，**执行这些命令所需的所有参数被封装在一个称为帧的数据结构中**，帧对数据进行编码以便传输。帧为命令及其参数提供了一种有效的方式，用于在网络上进行编码和分隔。
- 低层AMQP帧由五个不同的组件组成：
    1. 帧头
        1. 帧类型（五种）
        2. 信道编号
        3. 以字节为单位的帧大小
    2. 帧有效载荷
    3. 结束字节标记（ASCII值206）
- 注意
    - AMQP 中的**心跳行为**用于确保客户端和服务器之间相互响应，这是展示 AMQP 作为一种**双向 RPC 协议**的完美示例。可以通过更改 rabbitmq.config 文件中的 heartbat 值来更改 RabbitMQ 的最大心跳间隔
- 五种帧类型
    - **协议头帧**：用于连接到 RabbitMQ，仅使用一次。（对开发者是透明的）
    - **方法帧**：携带发送给 RabbitMQ 或从 RabbitMQ 接收到的 RPC 请求或响应。（可见的）
    - **内容头帧**：包含一条消息的大小和属性。（可见的）
    - **消息体帧**：包含消息的内容。（可见的）
    - **心跳帧**：在客户端与RabbitMQ之间进行传递，作为一种校验机制确保连接的两端都可用且在正常工作。（对开发者是透明的）

### 2.2.2 将消息编组成帧（方法帧==》内容头帧==》消息体帧）

- 使用方法帧、内容头帧和消息体帧向 RabbitMQ 发布消息。
- 发送的第一个帧是携带命令和执行它所需参数（如交换器和路由键）的方法帧。方法帧之后是内容帧，包含内容头和消息体。内容头帧包含消息属性以及消息体大小。
- **AMQP 的帧大小有一个上限，如果消息体超过这个上限，消息内容将被拆分成多个消息体帧。**
- 当向 RabbitMQ 发送消息时，**在方法（Method）帧中会发送一个 Basic.Publish 命令**，随后是一个带有消息属性的**内容头（ContentHeader）帧**，该内容头帧包括消息的内容类型和发送时间

等。这些属性被封装在AMQP规范中所定义的 **Basic.Properties** 数据结构中。最后，消息内容被编组到一定数量的**消息体帧**中。

- 方法帧和内容头帧：二进制打包数据。消息体帧：任何数据。

### 2.2.3 方法帧、内容头帧、消息体帧

- 方法帧结构
  - 方法帧携带着构建RPC请求所需的类、方法以及相关参数。
  - Basic.Publish 方法帧由五个组件组成：**用于标识 Basic.Publish RPC 请求的类和方法类型、交换器名称、路由键值和mandatory标志。**
  - 方法帧有效载荷中的**每个数据值都是按照数据类型采用特定的格式进行编码。**这种编码格式旨在最大限度地减少网络传输中的字节大小，提供数据完整性，并确保数据编组和解组速度尽可能快。实际的格式根据数据类型的不同而不同，但通常表现为一个字节后跟数字数据，或者是一个字节后跟一个字节大小的字段，其后则是文本数据。
- 内容头帧
  - 在方法帧之后发送的消息头除了告知 RabbitMQ 该**消息的大小**之外，消息头帧还包含**消息的各种属性**，为 RabbitMQ 服务器和可能接收它的任何应用程序提供了对消息的描述。这些**属性存储在 Basic.Properties 映射表**中，可能包含描述消息内容的数据，也可能是完全空白。大多数客户端库将预先填充一小部分字段，比如内容类型和投递模式。
- 消息体帧
  - 消息的消息体帧与正在传输的数据类型无关，并且可能包含二进制或文本数据。消息体帧都是消息中包含实际消息数据的结构。
  - 消息属性和消息体组合在一起构成了数据的强大封装格式。将消息的描述性属性与内容无关的消息体结合起来，确保你可以使用RabbitMQ来处理你认为合适的任何类型的数据。

# 三、RabbitMQ 配置

- 环境变量（官方文档）：
  - rabbitmq-env.conf(linux)
  - rabbitmq-env-conf.bat(Windows)
- rabbitmqctl：管理 vhost、用户和权限；管理 **运行参数和策略**
- rabbitmq-queues：管理特定于仲裁队列（quorum queue）的设置的工具
- rabbitmq-plugins：管理 plugin
- rabbitmq-diagnostice：允许检查**节点状态**，包括**有效配置**，以及许多其他指标和运行状况检查。
- 参数与策略（policy）：定义了可以在**运行时更改的集群范围的设置**，以及可以方便地为队列组（交换等）配置的设置，例如包括可选的队列参数。
- Runtime（Erlang VM）Flags：控制系统的底层方面：**内存分配设置、节点间通信缓冲区大小，运行时调度程序设置等**

## 3.1 配置文件

- （包括 server(核心服务器) 和 plugin 的设置：）配置文件地址
  - rabbitmq.conf（从 rabbitmq 3.7.0开始，配置格式使用 sysctl format 的格式）
    - 参数配置
    - 语法：
      - One setting uses one line（一个配置一行）
      - Lines are structured Key = Value（key/value 的方式）
      - Any line starting with a # character is a comment（以 # 开头的是注释）

    - advanced.config
        - 不能用 sysctl 格式来配置的参数
        - 类似于 rabbitmq.config
    - rabbitmq.conf 和 advanced.config 的改变在 RabbitMQ 节点重启后生效
    - RabbitMQ 3.7.0 之前的配置文件为：rabbitmq.config
    - **核心服务器参数设置（官网文档）**
    - **部分 plugins 参数设置（官网文档）**
- **配置值加密**：遵从一些必要的规范，让些敏感的数据不会出现在文本形式的配置文件中。
    - 在配置文件中将加密之后的值以“{encrypted ， 加密的值}”的形式包裹
- **操作系统内核限制**：虚拟内存，堆栈大小，打开的文件句柄等等
    - 修改限制：
        - 使用 systemd
        - 使用 Docker
        - 没有 systemd（旧的Linux发行版）： vim /etc/default/rabbitmq-server 或 rabbitmq-env.conf

## 3.2 修改配置的四种方式

1. 修改 rabbitmq.config 配置文件
2. 通过 rabbitmqctl 工具修改配置
3. 通过 RabbitMQ Management 插件提供的 HTTPAPI 接口来修改配置
4. 添加 Policy 修改队列的属性参数。（Policy 通常用来配置 Federation、镜像队列、备份交换器、死信等功能）

# 四、RabbitMQ 网络

## 4.1 使用 Heartbeat 和 TCP keepalive 检测无效的 TCP 连接

- 功能：
    - 确保应用程序层迅速发现中断的连接（以及完全不响应的对等点）
    - 心跳还可以防御某些网络设备，这些设备可能会在一段时间内没有活动时终止“空闲” TCP连接。
- 操作系统检测到 TCP 断开是一个适中的时间（在 Linux 中默认时长是 11 分钟）。AMQP 0-9-1 提供心跳检测功能来确保应用层及时发现中断的连接（或者是完全没有工作的连接）。 心跳检测还能保护连接不会在一段时间内没有活动而被终止。
- 心跳超时间隔
    - heartbeat timeout 值决定了 TCP 相互连接的最大时间，超过这个时间，该连接将被 RabbitMQ 和 客户端当作不可到达。这个值是**在 RabbitMQ 服务器和客户端连接的时候协商**的。客户端需要配置请求心跳检测。RabbotMQ 3.0 及以上的版本中，RabbitMQ 默认设置与客户端之间的心跳时长为 60 秒。
    - 心跳帧每隔 timeout/2 时间会发送一次。**连续两次心跳失败后，连接将会当作不可到达。**不同客户端对此的表现不同，但是 TCP 连接都会关闭。当客户端检测到 RabbitMQ 服务节点不可到达，它需要重新发起连接。
    - 任何连接数据交换（例如 协议操作、发布消息、消息确认）都会计入有效的心跳。客户端可能也会发送心跳包，在连接中有其他数据交换，但有些只在需要时发送心跳包。
- 禁用心跳：
    - 设置超时时间为 0 或设置很高的数（如 1800），因为帧传输太少，无法产生实际的变化。
    - 除非使用 TCP keepalive，否则**不建议禁用 Heartbeat**

    - 禁用了 Heartbeat，将很难及时发现对等方不可用，这将对数据安全构成重大风险
- 将 heartbeat timeout 值设置得太低，可能会导致由于网络瞬态拥塞，服务器流量控制寿命短暂等原因导致误报。对于大多数环境，**5到20秒范围内的值是最佳的。**
- Heartbeat 和 TCP 代理

    > - 当某些网络工具（HAproxy，AWS ELB）和设备（硬件负载平衡器）在一段时间内没有任何活动时，可能会终止“空闲” TCP 连接。在大多数情况下，这是不可取的。
    > - 在连接上启用心跳后，将导致周期性的轻型网络流量。因此，**心跳具有保护客户端连接的副作用，该客户端连接可能会闲置一段时间，以防止代理和负载平衡器过早关闭。**
    > - 心跳超时为 30 秒时，连接将大约每 15 秒产生一次周期性的网络流量。5到15秒范围内的活动足以满足大多数流行的代理和负载平衡器的默认设置。另请参见上面有关低超时和误报的部分。

- TCP keepalives：具有类似用途，但需要内核调整
- 对活动连接和断开连接进行故障排除：
    - RabbitMQ 节点将记录由于缺少心跳而关闭的连接。
    - 检查服务器和客户端日志将提供有价值的信息，并且应该是故障排除的第一步。
- Heartbeat：
    - 使用 Java 客户端启用 Heartbeat

        ```
        ConnectionFactory cf = new ConnectionFactory();

        // set the heartbeat timeout to 60 seconds
        cf.setRequestedHeartbeat(60);
        ```

        - 如果服务端设置了一个不为 0 的值，客户端只能低于这个值。

## 4.2 网络分区（脑裂问题）

- 当一个集群发生网络分区时，这个集群会分成两个部分或者更多，它们各自为政，互相都认为对方分区内的节点已经挂了，包括队列、交换器及绑定等元数据的创建和销毁都处于自身分区内，与其他分区无关。
- 网络分区带来的影响大多是负面的，极端情况下不仅会造成数据丢失，还会影响服务的可用性。
- 出现情况，与 net_ticktime(默认 60s) 这个参数息息相关的
    - 网络设备（比如中继设备、网卡）出现故障也会导致网络分区。
    - 如果一个节点不能在 T（取值范围: 0.75 × net_ticktime < T < 1.25 × net_ticktime） 时间连上另一个节点，那么 Mnesia 通常认为这个节点已经挂了，就算之后两个节点又重新恢复了内部通信，但是这两个节点都会认为对方已经挂了， Mnesia 此时认定了发生网络分区的情况。
- RabbitMQ 为什么还要引入网络分区?
    - 其中一个原因就与它本身的**数据一致性复制原理**有关。
- **查看是否出现网络分区的四种方式**：
    - RabbitMQ 服务日志：出现类似 ** ERROR ** mnesia event got 日志，则出现网络分区。
    - rabbitmqctl cluster_status 命令：在 partitions 一项中有相关内容，则出现网络分区。
    - 通过 Web 管理界面的方式查看：出现网络分区探测警告，则出现网络分区。
    - 通过 HTTP API 的方式调取节点信息来检测是否发生网络分区：partitions 项有内容，则出现网络分区。
- 手动从网络分区中恢复：
    - 挑选一个信任区：有决定 Mnesia 内容的权限。
    - 3 个要点：

1. 如何挑选信任分区？（**优先级从上到下**）
    1. 分区中要有 disc 节点。
    2. 分区中的节点数最多；
    3. 分区中的队列数最多；
    4. 分区中的客户端连接数最多。
2. 如何重启节点？（**两种重启方式**）
    1. 使用 rabbitmqctl stop 命令关闭，然后再用 rabbitmq-server -detached 命令启动（同时重启 Erlang 虚拟机和 RabbitMQ 应用）
    2. 使用 rabbitmqctl_stop_app 关闭，然后使用 rabbitmqctl_start_app 命令启动（**只需要重启 RabbitMQ 应用，推荐**）
3. 重启的顺序有何考究？（**两种重启顺序中选一个**）
    1. **停止其他非信任分区中的所有节点，然后再启动这些节点。**如果此时还有网络分区的告警，则再重启信任分区中的节点以去除告警。
    2. 关闭整个集群中的节点，然后再启动每一个节点，这里需要确保启动的第一个节点在信任的分区之中。
    3. 考虑队列"漂移"现象（只出现在镜像队列中）：master 漂移（关闭重启后所有 master 漂移到同一个节点上，不能实现负载均衡。）
        1. 重启前删除镜像队列的配置，一定程度上缓解 master 漂移。（rabbitmqctl / Web 管理界面 / HTTP API 删除）

- **具体的网络分区恢复步骤：**
    1. 挂起生产者和消费者进程。这样可以减少消息不必要的丢失，如果进程数过多，情形又比较紧急，也可跳过此步骤。
    2. 删除镜像队列的配置。
    3. 挑选信任分区。
    4. 关闭非信任分区中的节点。采用 rabbitmqctl stop_app 命令关闭
    5. 启动非信任分区中的节点。采用与步骤 4 对应的 rabbitmqctl start_app 命令启动。
    6. 检查网络分区是否恢复，如果已经恢复则转步骤 8； 如果还有网络分区的报警则转步骤 7。
    7. 重启信任分区中的节点。
    8. 添加镜像队列的配置。
    9. 恢复生产者和消费者的进程。

- 自动从网络分区恢复
    - RabbitMQ 提供了三种方式：（默认不自动处理）
        1. pause-minority 模式
            1. 当发生网络分区时，自动检测自身是否处于“少数派”，RabbitMQ 会自动关闭（RabbitMQ 应用）这些节点的运作。（保障了 CAP 中的 P）
            2. 需要考虑出现 2v2 、3v3 等分裂成对等点数的分区情况。
            3. 适用情况：对于对等分区的处理不够优雅，可能会关闭所有的节点。一般情况下，可应用于非跨机架、奇数节点数的集群中。
        2. pause-if-all-down 模式
            1. 该模式下，RabbitMQ 集群中的节点在和所配置的列表中的任何节点不能交互时才会关闭。
            2. 适用情况：对于受信节点的选择尤为考究，尤其是在集群中所有节点硬件配置相同的情况下。此种模式可以处理对等分区的情形。
        3. autoheal 模式。
            1. 在 autoheal 模式下，当认为发生网络分区时 RabbitMQ 会自动决定一个获胜（winning）的分区，然后重启不在这个分区中的节点来从网络分区中恢复。

2. 适用情况：可以处于各个情形下的网络分区 。但是如果集群中有节点处于非运行状态，则此种模式会失效。

# 五、RabbitMQ 运维

## 5.1 查看服务日志

- RabbitMQ 的日志默认存放在 /var/log/rabbitmq（或者 /data/rabbitmq/log） 文件夹内。在这个文件夹内 rabbitMQ 会创建两个日志文件
  - RABBITMQ_NODENAME-sasl.log：Erlang 的异常日志
  - RABBITMQ_NODENAME.log：RabbitMQ 应用服务日志

## 5.2 内存和磁盘告警

- 生产者和消费者不应该设置在同一个 Connection 上，这样发生阻塞时可以在阻止生产者的同时而又不影响消费者的运行。

### 5.2.1 内存告警

- 当 RabbitMQ 使用的内存超过内存阈值时，就会产生内存告警并阻塞所有生产者的连接。一旦告警被解除（有消息被消费或者从内存转储到磁盘等情况的发生），一切都会恢复正常。
- 内存阈值设置方式：
  - rabbitmqctl set_vm_memory_high_watermark {fraction}
  - 通过 rabbitmq.config 配置文件来配置
- 查看服务日志（/data/rabbitmq/log/）

### 5.2.2 磁盘告警

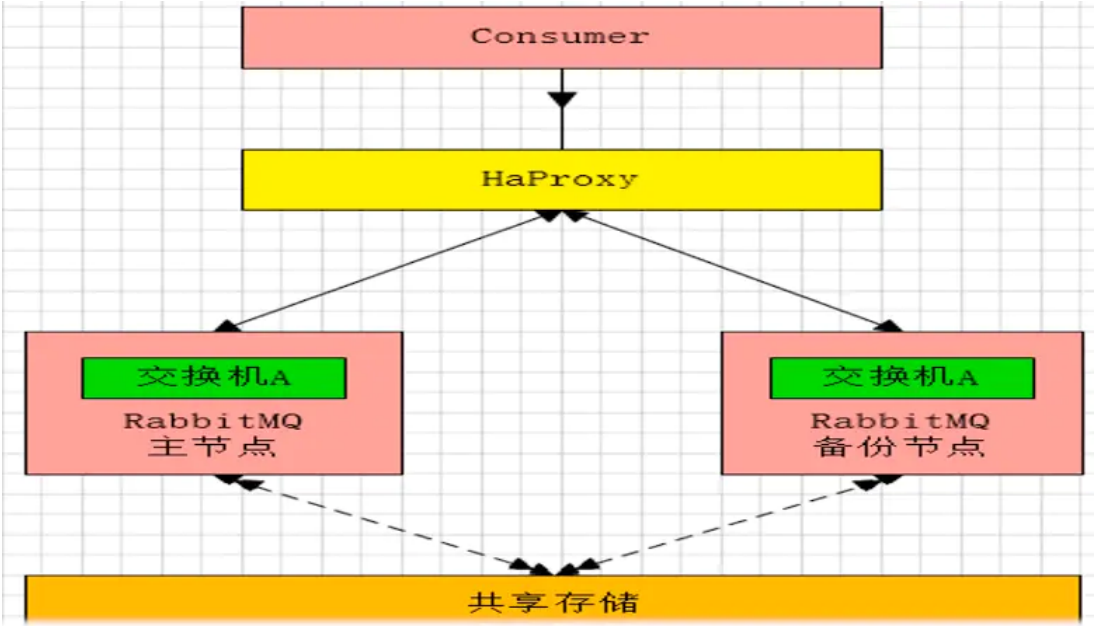
- 当磁盘空间低于确定的阈值（默认为 50 MB）时，RabbitMQ 会阻塞生产者，避免因非持久化消息的持续换页而耗尽磁盘空间导致服务崩溃。
- 相对谨慎的做法是将磁盘阈值设置为与操作系统所显示的内存大小一致。
- 设置方式
  - rabbitmqctl set_disk_free_limit {disk_limit}
  - 通过 rabbitmq.config 配置文件来配置

# 六、RabbitMQ 高可用

## 6.1 RabbitMQ 的4种集群架构

### 6.1.1 主备模式

- 实现 rabbitMQ 的高可用集群，一般**在并发和数据量不高的情况下，**这种模式非常的好用且简单。
- 原理：就是一个主/备方案，主节点提供读写，备用节点不提供读写。如果主节点挂了，就切换到备用节点，原来的备用节点升级为主节点提供读写服务，当原来的主节点恢复运行后，原来的主节点就变成备用节点，和 activeMQ 利用 zookeeper 做主/备一样，也可以一主多备。




### 6.1.2 远程模式（shovel）

- 远程模式可以实现双活的一种模式，简称 shovel 模式，所谓的 shovel 就是把消息进行不同数据中心的复制工作，可以跨地域的让两个 MQ 集群互联，远距离通信和复制。 Shovel 就是我们可以把消息进行数据中心的复制工作，我们可以跨地域的让两个 MQ 集群互联。
- 在使用了 shovel 插件后，模型变成了近端同步确认，远端异步确认的方式，大大提高了订单确认速度，并且还能保证可靠性。

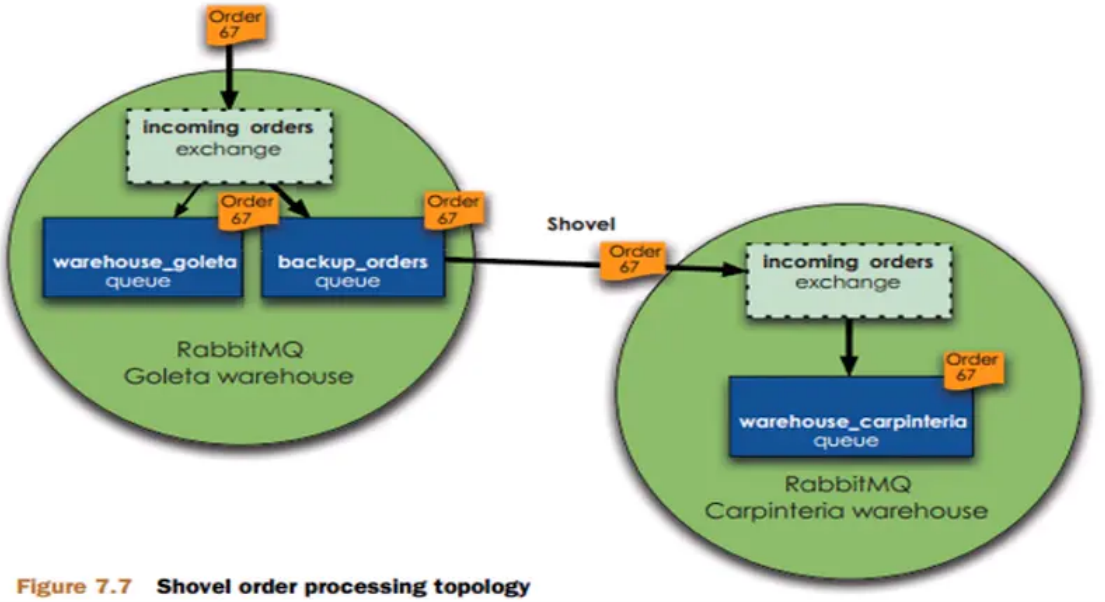
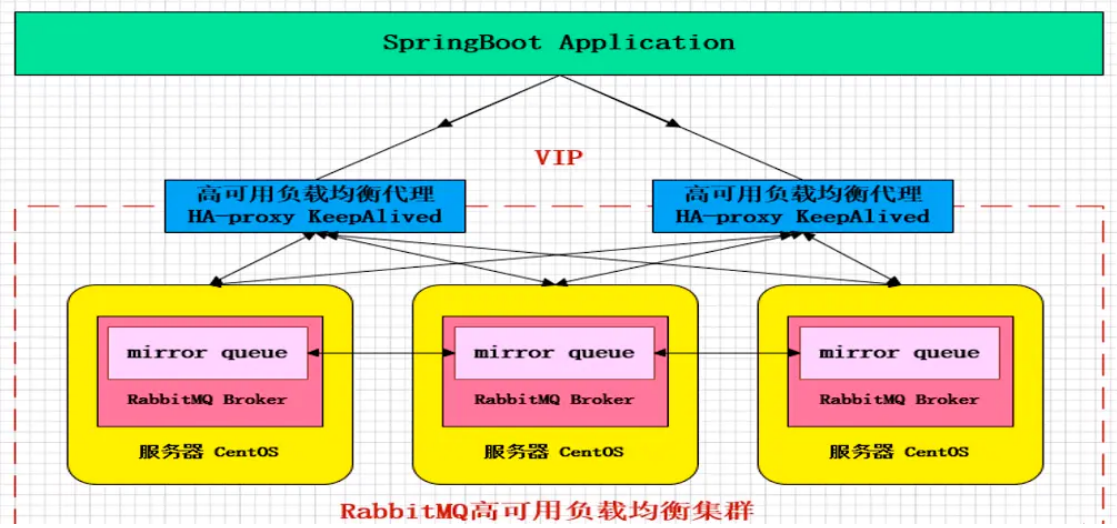



Figure 7.7 **Shovel order processing topology**

- 如上图所示，当我们的消息到达 exchange，它会判断当前的负载情况以及设定的阈值，如果负载不高就把消息放到我们正常的 warehouse_goleta 队列中，如果负载过高了，就会放到 backup_orders 队列中。backup_orders 队列通过 shovel 插件与另外的 MQ 集群进行同步数据，把消息发到第二个 MQ 集群上。

### 6.1.3 镜像队列模式

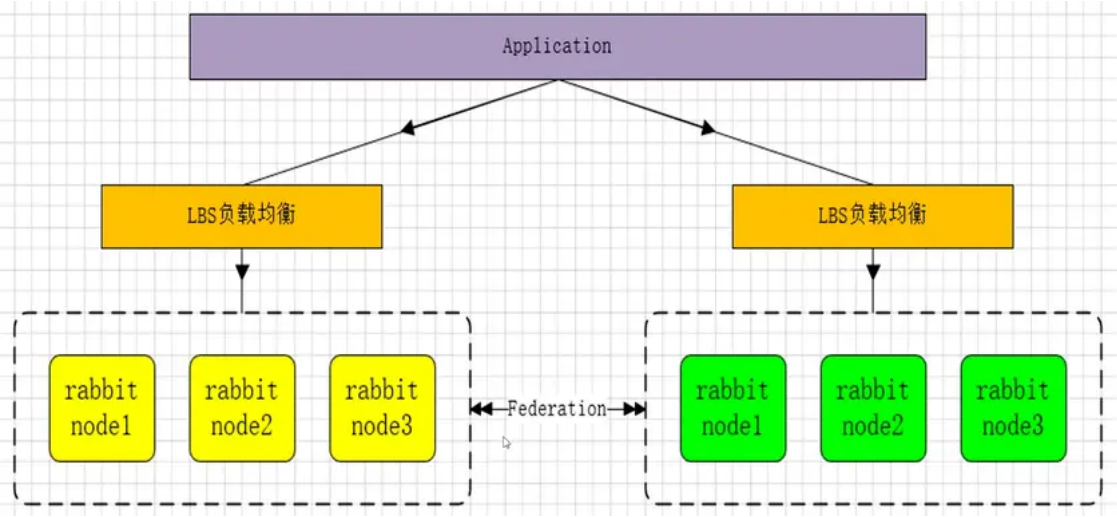
- 非常经典的 mirror 镜像模式，**保证 100% 数据不丢失**。在实际工作中也是用得最多的，并且实现非常的简单，一般互联网大厂都会构建这种镜像集群模式。mirror 镜像队列，目的是为了保证 rabbitMQ 数据的高可靠性解决方案，主要就是实现数据的同步，一般来讲是 2 - 3 个节点实现数据同步。对于 100% 数据可靠性解决方案，一般是采用 3 个节点。
- 集群架构如下：




- **镜像是按队列来算的，而不是按节点来算的。Mirror queue。**

### 6.1.4 多活模式（federation）

- 也是实现异地数据复制的主流模式，因为 shovel 模式配置比较复杂，所以一般来说，**实现异地集群的都是采用这种双活 或者 多活模型来实现的。**这种模式需要依赖 rabbitMQ 的 federation 插件，可以实现持续的，可靠的 AMQP 数据通信，多活模式在实际配置与应用非常的简单。
- rabbitMQ 部署架构采用双中心模式(多中心)，那么在两套(或多套)数据中心各部署一套 rabbitMQ 集群，各中心的rabbitMQ 服务除了需要为业务提供正常的消息服务外，中心之间还需要实现部分队列消息共享。
- 多活集群架构如下：




## 6.2 高可用

- 搭建镜像队列集群
- 使用 HAProxy 实现镜像队列的负载均衡
- 使用 KeepAlived 搭建高可用的 HAProxy 集群

# 七、RabbitMQ 原理

## 7.1 存储机制

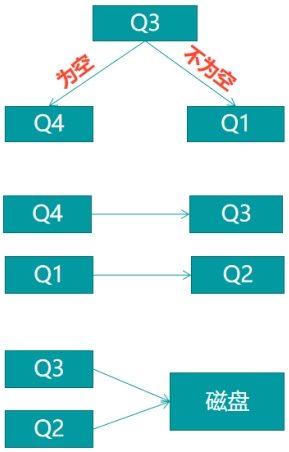
- 持久化和非持久化消息（都在“持久层”处理完成）
  - 持久化消息在到达队列时就被写入到磁盘，如果内存足够，还会在内存中保存一份备份，提高一定的性能，当内存吃紧的时候从内存中清除。
  - 非持久化消息一般只保存在内存中，在内存吃紧的时候会被换入到磁盘空间，以节省内存空间。
- 持久层时逻辑概念，实际上包含两个部分：
  - **队列索引（rabbit_queue_index)**：负责维护队列中落盘消息的信息，包括消息的存储地点、是否已被交付给消费者、是否已被消费者 ack 等。每个队列都有与之对应的一个 rabbit_queue_index。
  - **消息存储（rabbit_msg_store)**：以键值对的形式存储消息，它被所有队列共享，在每个节点中有且只有一个。从技术上可分为两种：
    - msg_store_persistent：负责持久化消息的持久化，重启后消息不会丢失；
    - msg_store_transient：负责非持久化消息的持久化，重启后消息会丢失。
  - 消息（包括消息体、属性和 headers ）可以直接存储在 rabbit_queue_index 中，也可以被保存在 rabbit_msg_store 中。
- 存储文件目录：msg_store_persistent、msg_store_transient、queues
- **最佳的配备**：是较小的消息存储在 rabbit_queue_index 中而较大的消息存储在 rabbit_msg_store 中
- 消息的读取、删除（标记为垃圾数据）、垃圾回收（文件合并）

### 7.1.1 队列的结构

- 由以下两部分组成
  - rabbit_amqqueue_process：负责协议相关的消息处理，即接收生产者发布的消息、向消费者交付消息、处理消息的确认（包括生产端的 confirm 和消费端的 ack ）等。
  - backing_queue：是消息存储的具体形式和引擎，并向 rabbit_amqqueue_process 提供相关的接口以供调用。
- 如果消息投递的目的队列为空，且由消费者订阅了这个队列，那么该消息会直接发送给消费者，不会经过队列这一步。
- RabbitMQ 中的队列消息可能会处于以下**四种状态**：
  1. alpha：消息内容（包括消息体、属性和 headers）和消息索引都存储在内存中。
  2. beta：消息内容保存在磁盘中，消息索引保存在内存中。
  3. gamma：消息内容保存在磁盘中，消息索引在磁盘和内存中都有。（只有持久化的消息才有的状态）
  4. delta：消息内容和索引都在磁盘中。
- 区分这四种状态的主要作用是满足不同的内存和 CPU 需求:
  - alpha：最耗内存，但很少消耗 CPU
  - delta：需要执行两次 I/O 操作才能读取到消息，一次是读取消息索引（从 rabbit_queue_index 中），一次是读取消息内容（从 rabbit_msg_store 中）
  - beta 和 gamma 状态都只需要执行一次 I/O 操作就可以读取到消息 （从 rabbit_msg_store 中）
- 对于普通的没有设置优先级和镜像的队列，backing_queue 的默认实现是 rabbit_variable_queue，其内部通过 5 个子队列 Q1、Q2、Delta、Q3 和 Q4 来体现消息的各个状态。
  - 消息流动顺序（不一定经历所有状态）：Q1 ===》 Q2 ====》 Delta ===》 Q3 ===》 Q4
  - 从 Q1 到 Q4 基本经历：内存到磁盘，再由磁盘到内存这样的一个过程，如此可以在队列负载很高的情况下，能够通过将部分消息由磁盘保存来节省内存空间，而在负载降低的时候，

这部分消息又渐渐回到内存被消费者获取，使得整个队列具有很好的弹性。

**写到内存队列中**



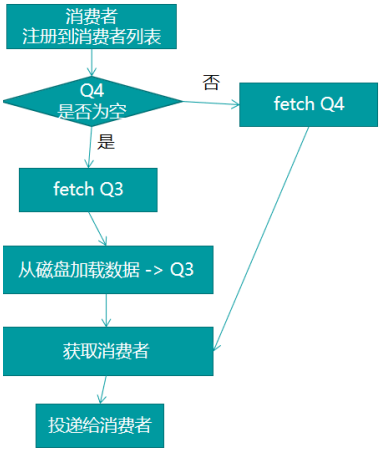

```
if len(Q3)==0:
  Q4.append(data)
else:
  Q1.append(data)

if RamMsgCount+ram_pending_ack - TargetRamCount >0:
  limit_ram_acks() #把待ack的先持久化
  #把alphas状态的数据转到beta
  #Q1->Q2, Q4->Q3
  push_alphas_to_betas()

if len(Q2)+len(Q3)- permitted_beta_count  >= 2048:
  #Q2,Q3 -> 磁盘
  push_betas_to_deltas()
```

- 消费者获取消息也会引起消息的状态转换。

**消息消费过程**




```
if len(Q4)==0:
  if len(Q3)==0:
      return;
  msg = Q3.pop()
  if len(Q3)==0:
      #通过最小的已持久化到磁盘的SeqId去加载数据
      Q3.append(queue_index.read(DeltaSeqId))
else:
  msg = Q4.pop()

consumerId = getConsumer()
rabbit_queue_consumers:deliver(msg,consumerId)
```

### 7.1.2 惰性队列

- 惰性队列会尽可能地将消息存入磁盘中，而在消费者消费到相应的消息时才会被加载到内存中，它的一个重要的设计目标是**能够支持更长的队列，即支持更多的消息存储。**当消费者由于各种各样的原因（比如消费者下线、宕机或者由于维护而关闭等）致使长时间内不能消费消息而造成堆积时，惰性队列就很有必要了。
- 惰性队列会将接收到的消息直接存入文件系统中，而不管是持久化的或者是非持久化的，这样可以减少了内存的消耗，但是会增加 I/O 的使用，如果消息是持久化的，那么这样的 I/O 操作不可避免，**惰性队列和持久化的消息可谓是“最佳拍档”。**
- 队列具备两种模式：default 和 lazy，默认是 default。
- 惰性队列和普通队列相比，只有很小的内存开销。

## 7.2 流控

- **流控机制是用来避免消息的发送速率过快而导致服务器难以支撑的情形。**
  - 内存和磁盘的告警相当于全局的流控（Global Flow Control），一旦触发会阻塞集群中所有的 Connection。
  - 本次的流控机制时针对单个 connection 的，可称为 Per-Connection Flow Control 或 Internal Flow Control
- 原理：
  - 当有大量消息发往某个进程时，会导致该进程邮箱（mailbox）过大，最终内存溢出崩溃。

    - RabbitMQ 使用了一种基于**信用证算法（ credit-based algorithm ）的流控机制**来限制发送消息的速率以解决前面所提出的问题。
    - 基于信用证的流控机制最终将消息发送进程的发送速率限制在消息处理进程的处理能力范围之内。
- 流控机制不只是作用于 Connection ，同样作用于信道（channel) 和队列。**从 Connection 到 Channel，再到队列，最后是消息持久化存储形成一个完整的流控链**，对于处于整个流控链中的任意进程，只要该进程阻塞，上游的进程必定全部被阻塞。也就是说，如果某个进程达到性能瓶颈，必然会导致上游所有的进程被阻塞。所以我们可以利用流控机制的这个特点找出瓶颈之所在。
- **几个关键进程及对应顺序**：
    1. rabbit_reader（connection）：Connection 的处理进程，负责接收、解析 AMQP 协议数据包等。
    2. rabbit_channel：Channel 的处理进程，负责处理 AMQP 协议的各种方法、进行路由解析等。
    3. rabbit_amqqueue_process：队列的处理进程，负责实现队列的所有逻辑。
    4. rabbit_msg_store：负责实现消息的持久化。
- 打破队列瓶颈的案例——通过封装将多个队列连接成一个队列。

# 八、RabbitMQ 管理系统业务需求

## 8.1 消息队列

- 申请：需要审核人审核的申请都发送 VV 提醒。
- 用户绑定产品申请：选择绑定的产品，提交申请，由审核人在即云上审核。
- Vhost 申请
    - 填写名称、备注，选择集群、所属产品，然后提交，即可申请成功。
- vhost 列表
    - 添加或修改 vhost
    - nodes：
        - 读写测试：测试 RabbitMQ 是否通讯正常
        - 删除：删除所有节点上的 vhost：MQ_Test
    - pro_name：修改关联产品（vhost 消费方）
    - op（options）：
        - 队列列表：添加或修改队列（Vhost、Exchange、queueName、routingKey、durable、autoDelete、Notes(备注)、Nodes(只能选择 vhost 属于的节点)、消费方所属产品）
        - 交换机列表：添加 Exchange
            - Policies：添加对应 Exchange 的 Policy，修改指定参数
        - Policies：添加 Policy，修改指定参数（可以选择 Exchange 和 apply-to）
            - apply-to：exchanges and queues、exchanges、queues
        - 修改最大队列数量（默认 200）
        - 修改/删除：修改/删除 Vhost
- queue 列表（队列管理）
    - 生产法：设置可以下线（生产方和消费方都设置下线后才可以删除队列）
    - 消费方：
        - 修改：更改消费方
        - 设置可以下线：同上

    - detail
        - 点击节点名称按钮：查看节点下的**数据堆积**和**写入和消费**
            - **数据堆积**和**写入和消费** 曲线图（total、Ready、Unack）
            - **Consumers list**
            - **Shovel list**
            - **Binding**
            - **Detail  node**
        - 删除：删除所有节点上的该队列
    - op（options）
        - 报警：添加报警用户列表（结合RMS）
        - 新增同步：数据同步（将指定队列数据同步到另一队列中(可以指定节点、vhost、队列)）
        - 同步列表：查看 shovel 同步列表
        - Policies：添加对应 queue 的 Policy，修改指定参数
        - 报警策略：默认使用全局报警策略，可以自己添加相关的报警策略
- 流量监控（堆积）——流量趋势图（堆积）——一天以内
    - 查看所有节点的流量趋势图（堆积）——可选（节点、开始时间、结束时间）
        - messages
        - message_ready
        - connectionTotal
        - messages_unack
        - child_node_count
- RabbitMQ 用户管理（给 RabbitMQ 账号授权）
    - 添加 RabbitMQ 连接帐号（用户名、密码、Notes、Nodes(可多选)、所属产品）
    - limit（账号下的限流）：
        - 修改：更改 账号下的 限流限制
    - 绑定产品：
        - 修改：更改该账号和产品的绑定
    - op（options）
        - 修改：修改该账号的密码、备注、Nodes、所属产品
        - **授权**：
            - 查看全部：显示所有队列（已授权和未授权），进行选择授权
            - 已有权限：隐藏未授权的队列
            - 下来列表（选择账号）：查看该账号下的队列
            - 授权：
                - 读取
                - 写入
            - 生成配置
        - 删除：删除指定账户
- 队列同步搜索列表（shovel列表）
    - 状态（state）：正在同步、暂定
    - 批量操作：暂定、恢复、删除
- 写入消费流量监控获取：查看流量写入、消费的情况

    - 节点、时间、Vhost、队列、取样时间间隔
- 集群列表
    - 监控得分：一项不达标扣十分
    - 监控指标：监控阈值，可修改
- 队列流量统计（按天）
    - 获取队列的写入和消费数据、用于服务计费

## 8.2 用户中心

- 产品流量统计
    - 查看产品的写入数据、消费数据、写入队列个数、消费队列个数，用于服务计费
    - 可查看明细
- 操作日志
- 产品管理
    - 更改产品的**默认报警配置**
- 用户列表
    - 修改用户：用户姓名、分配用户组（角色）、绑定产品
    - 删除
- 用户组（角色）列表
    - 新增
        - 类型、名称、备注
    - 授权：分配 api 接口权限

# 九、常见问题

## 9.1 节点启动问题

- 通常情况下，**当关闭整个 RabbitMQ 集群时，重启的第一个节点应该是最后关闭的节点**，因为它可以看到其他节点所看不到的事情。但是有时会有一些异常情况出现，比如整个集群都断电而所有节点都认为它不是最后个关闭的。在这种情况下，可以调用 `rabbitmqctl force_boot` 命令，这就告诉节点可以无条件地启动节点 。在此节点关闭后，集群的任何变化，它都会丢失。如果最后一个关闭的节点永久丢失了，那么需要优先使用 rabbitmqctl forget cluster_node {nodename} 命令，因为它可以确保镜像队列的正常运转。

## 9.2 单节点故障恢复

- 需要在集群中的其他节点中执行 rabbitmqctl forget_cluster_node {nodename} 命令来将故障节点剔除，其中 nodename 表示故障机器节点名称。如果之前有客户端连接到此故障节点上，在故障发生时会有异常报出，此时需要将故障节点的地址从连接列表里删除，并让客户端重新与集群中的节点建立连接，以恢复整个应用。

## 9.3 其他

- 产品没有设置默认 RMS 报警：根据报警的 vhost 查找产品，通知产品管理员设置 RMS 报警。