图灵：楼兰
你的神秘技术宝藏

这一章节作为课程收尾，给大家讨论一些在生产使用时需要注意的一些问题。

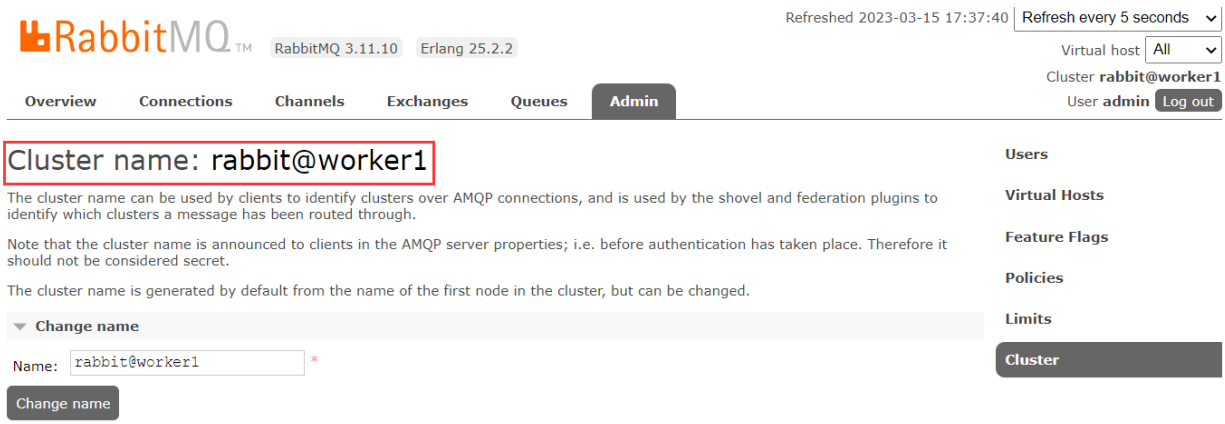
# 一、如何保证RabbitMQ服务高可用

## 1、RabbitMQ如何保证消息安全

之前通过单机环境搭建起来的RabbitMQ服务有一个致命的问题，那就是服务不稳定的问题。如果只是单机RabbitMQ的服务崩溃了，那还好，大不了重启下服务就是了。但是如果是服务器的磁盘出问题了，那问题就大了。因为消息都是存储在Queue里的，Queue坏了，意味着消息就丢失了。这在生产环境上肯定是无法接受的。而RabbitMQ的设计重点就是要保护消息的安全性。

所以RabbitMQ在设计之处其实就采用了集群模式来保护消息的安全。基础的思想就是给每个Queue提供几个备份。当某一个服务的Queue坏了，至少还可以从其他Queue中获取服务。

其实对于RabbitMQ，一个节点的服务也是作为一个集群来处理的，在web控制台的admin-> cluster 中可以看到集群的名字，并且可以在页面上修改。




那么RabbitMQ是怎么考虑数据安全这回事的呢？实际上，RabbitMQ考虑了两种集群模式：

- **默认的普通集群模式**：

  这种模式使用Erlang语言天生具备的集群方式搭建。这种集群模式下，集群的各个节点之间只会有相同的元数据，即队列结构，而消息不会进行冗余，只存在一个节点中。消费时，如果消费的不是存有数据的节点， RabbitMQ会临时在节点之间进行数据传输，将消息从存有数据的节点传输到消费的节点。

  很显然，这种集群模式的消息可靠性不是很高。因为如果其中有个节点服务宕机了，那这个节点上的数据就无法消费了，需要等到这个节点服务恢复后才能消费，而这时，消费者端已经消费过的消息就有可能给不了服务端正确应答，服务起来后，就会再次消费这些消息，造成这部分消息重复消费。 另外，如果消息没有做持久化，重启就消息就会丢失。

  并且，这种集群模式也不支持高可用，即当某一个节点服务挂了后，需要手动重启服务，才能保证这一部分消息能正常消费。

  所以这种集群模式只适合一些对消息安全性不是很高的场景。而在使用这种模式时，消费者应该尽量的连接上每一个节点，减少消息在集群中的传输。

- **镜像模式**：

  这种模式是在普通集群模式基础上的一种增强方案，这也就是RabbitMQ的官方HA高可用方案。需要在搭建了普通集群之后再补充搭建。其本质区别在于，这种模式会在镜像节点中间主动进行消息同步，而不是在客户端拉取消息时临时同步。

  并且在集群内部有一个算法会选举产生master和slave，当一个master挂了后，也会自动选出一个来。从而给整个集群提供高可用能力。

  这种模式的消息可靠性更高，因为每个节点上都存着全量的消息。而他的弊端也是明显的，集群内部的网络带宽会被这种同步通讯大量的消耗，进而降低整个集群的性能。这种模式下，队列数量最好不要过多。

# 2、搭建普通集群

接下来，我们准备三台服务器，在/etc/hosts文件中分配配置机器别名为worker1，worker2，worker3。然后三台服务器上分别搭建起RabbitMQ的服务，然后开始搭建集群。

1：需要同步集群节点中的cookie。

默认会在 /var/lib/rabbitmq/目录下生成一个.erlang.cookie。 里面有一个字符串。我们要做的就是保证集群中三个节点的这个cookie字符串一致。

我们实验中将worker1和worker3加入到worker2的RabbitMQ集群中，所以将worker2的.erlang.cookie文件分发到worker1和worker3。

> 同步文件时注意一下文件的权限，如果文件不可读，集群启动会有问题。简单粗暴的，可以使用chmod 777 .erlang.cookie指令给这个文件配置最大权限。

2：将worker1的服务加入到worker2的集群中。

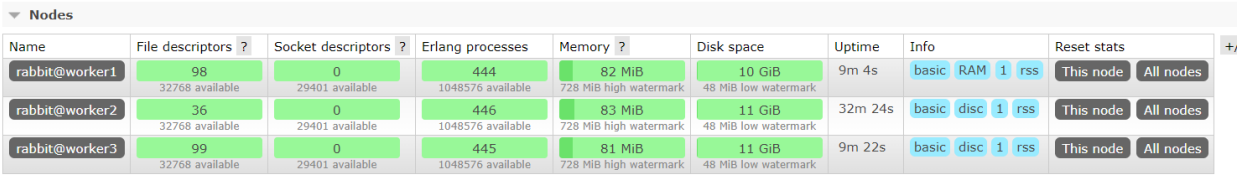
首先需要保证worker1上的rabbitmq服务是正常启动的。 然后执行以下指令：

```
[root@worker1 rabbitmq]# rabbitmqctl stop_app
Stopping rabbit application on node rabbit@worker1 ...
[root@worker1 rabbitmq]# rabbitmqctl join_cluster --ram rabbit@worker2
Clustering node rabbit@worker1 with rabbit@worker2
[root@worker1 rabbitmq]# rabbitmqctl start_app
Starting node rabbit@worker1 ...
```

> --ram 表示以Ram节点加入集群。RabbitMQ的集群节点分为disk和ram。disk节点会将元数据保存到硬盘当中，而ram节点只是在内存中保存元数据。
>
> 1、由于ram节点减少了很多与硬盘的交互，所以，ram节点的元数据使用性能会比较高。但是，同时，这也意味着元数据的安全性是不如disk节点的。在我们这个集群中，worker1和worker3都以ram节点的身份加入到worker2集群里，因此，是存在单点故障的。如果worker2节点服务崩溃，那么元数据就有可能丢失。在企业进行部署时，性能与安全性需要自己进行平衡。
>
> 2、这里说的元数据仅仅只包含交换机、队列等的定义，而不包含具体的消息。因此，ram节点的性能提升，仅仅体现在对元数据进行管理时，比如修改队列queue，交换机exchange，虚拟机vhosts等时，与消息的生产和消费速度无关。
>
> 3、如果一个集群中，全部都是ram节点，那么元数据就有可能丢失。这会造成集群停止之后就启动不起来了。RabbitMQ会尽量阻止创建一个全是ram节点的集群，但是并不能彻底阻止。所以，综合考虑，官方其实并不建议使用ram节点，更推荐保证集群中节点的资源投入，使用disk节点。

然后同样把worer3上的rabbitmq加入到worker2的集群中。

加入完成后，可以在worker2的Web管理界面上看到集群的节点情况：




> 也可以用后台指令查看集群状态 rabbitmqctl cluster_status

# 3、搭建镜像集群

这样就完成了普通集群的搭建。 再此基础上，可以继续搭建**镜像集群。**

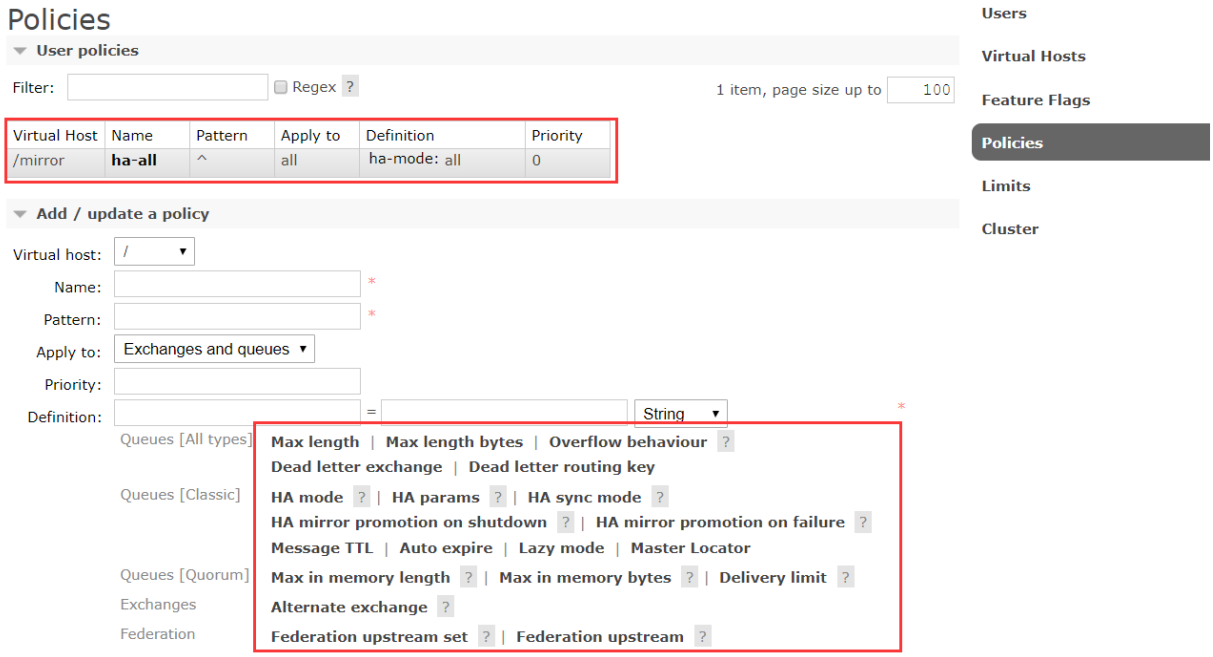
通常在生产环境中，为了减少RabbitMQ集群之间的数据传输，在配置镜像策略时，会针对固定的虚拟主机virtual host来配置。

> RabbitMQ中的vritual host可以类比为MySQL中的库，针对每个虚拟主机，可以配置不同的权限、策略等。并且不同虚拟主机之间的数据是相互隔离的。

我们首先创建一个/mirror的虚拟主机，然后再添加给对应的镜像策略：

```
[root@worker2 rabbitmq]# rabbitmqctl add_vhost /mirror
Adding vhost "/mirror" ...
[root@worker2 rabbitmq]# rabbitmqctl set_policy ha-all --vhost "/mirror" "^" '{"ha-mode":"all"}'
Setting policy "ha-all" for pattern "^" to "{"ha-mode":"all"}" with priority "0" for vhost "/mirror" ...
```

同样，这些配置的策略也可以在Web控制台操作。另外也提供了HTTP API来进行这些操作。




> 这些参数需要大致了解下。其中，pattern是队列的匹配规则, ^表示全部匹配。 ^ ha \ 这样的配置表示以ha开头。通常就用虚拟主机来区分就够了，这个队列匹配规则就配置成全匹配。
>
> 然后几个关键的参数：
>
> HA mode: 可选值 all，exactly，nodes。生产上通常为了保证高可用，就配all
>
> ```
> - all ： 队列镜像到集群中的所有节点。当新节点加入集群时，队列也会被镜像到这个节点。
> - exactly ： 需要搭配一个数字类型的参数(ha-params)。队列镜像到集群中指定数量的节点。如果集群内节点数少于这个数字，则队列镜像到集群内的所有节点。如果集群内节点少于这个数，当一个包含镜像的节点停止服务后，新的镜像就不会去另外找节点进行镜像备份了。
> - nodes: 需要搭配一个字符串类型的参数。将队列镜像到指定的节点上。如果指定的队列不在集群中，不会报错。当声明队列时，如果指定的所有镜像节点都不在线，那队列会被创建在发起声明的客户端节点上。
> ```
>
> 还有其他很多参数，可以后面慢慢再了解。

通常镜像模式的集群已经足够满足大部分的生产场景了。虽然他对系统资源消耗比较高，但是在生产环境中，系统的资源都是会做预留的，所以正常的使用是没有问题的。但是在做业务集成时，还是需要注意队列数量不宜过多，并且尽量不要让RabbitMQ产生大量的消息堆积。

> 这样搭建起来的RabbitMQ已经具备了集群特性，往任何一个节点上发送消息，消息都会及时同步到各个节点中。而在实际企业部署时，往往会以RabbitMQ的镜像队列作为基础，再增加一些运维手段，进一步提高集群的安全性和实用性。

> 例如，增加keepalived保证每个RabbitMQ的稳定性，当某一个节点上的RabbitMQ服务崩溃时，可以及时重新启动起来。另外，也可以增加HA-proxy来做前端的负载均衡，通过HA-proxy增加一个前端转发的虚拟节点，应用可以像使用一个单点服务一样使用一个RabbitMQ集群。这些运维方案我们就不做过多介绍了，有兴趣可以自己了解下。

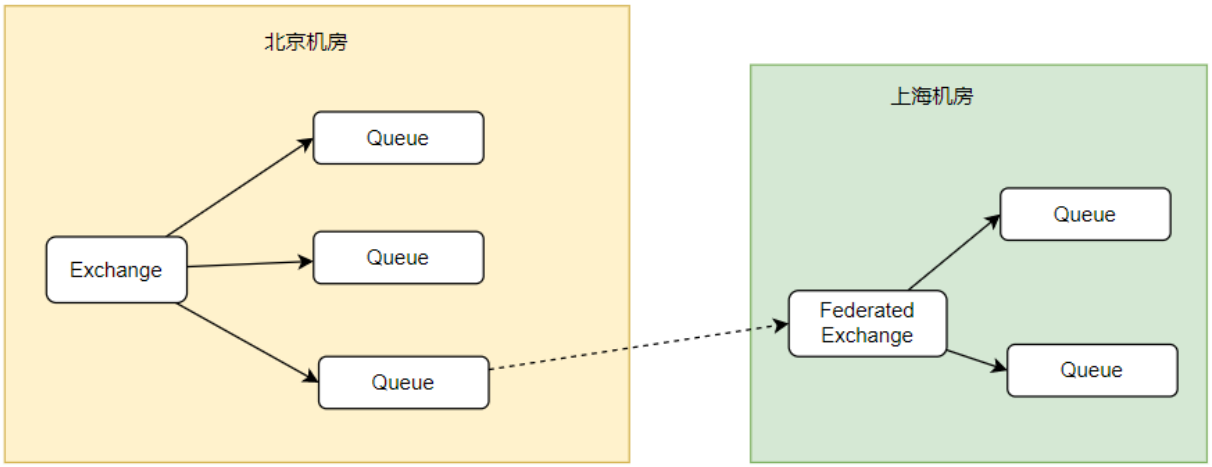
# 4、RabbitMQ常见的高可用集群部署方案--了解

有了镜像集群后，应用服务只需要访问RabbitMQ的集群中任意一个服务就可以了。但是在企业应用过程中，是不是有了镜像集群就够了呢？其实高可用这事还没有这么简单。

## 1、远程消息同步

镜像集群需要在集群内部进行频繁的数据同步，所以镜像集群有一个问题，就是对集群内的网络要求是挺高的。如果就在一个机房里面部署RabbitMQ镜像集群那没什么问题。 但是如果需要跨机房进行部署，这时候使用RabbitMQ镜像集群就不太合适了。

这时最好的方案是使用Federation联邦插件给关键的RabbitMQ服务搭建一个备份服务。



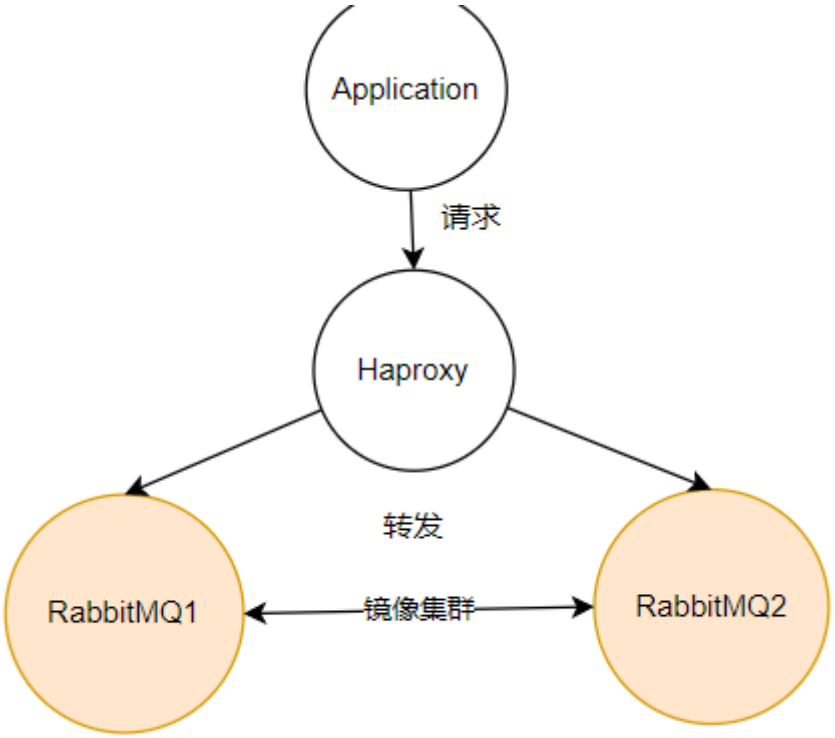

关于Federation插件的使用方式，在之前章节已经做过介绍，这里就不多做介绍了。

## 2、镜像集群+Haproxy+Keepalived

1、Haproxy反向代理

有了镜像集群之后，客户端应用就可以访问RabbitMQ集群中任意的一个节点了。但是，不管访问哪个服务，如果这个服务崩溃了，虽然RabbitMQ集群不会丢失消息，另一个服务也可以正常使用，但是客户端还是需要主动切换访问的服务地址。

为了防止这种情况情况，可以在RabbitMQ之前部署一个Haproxy，这是一个TCP负载均衡工具。应用程序只需要访问haproxy的服务端口，Haproxy会将请求以负载均衡的方式转发到后端的RabbitMQ服务上。



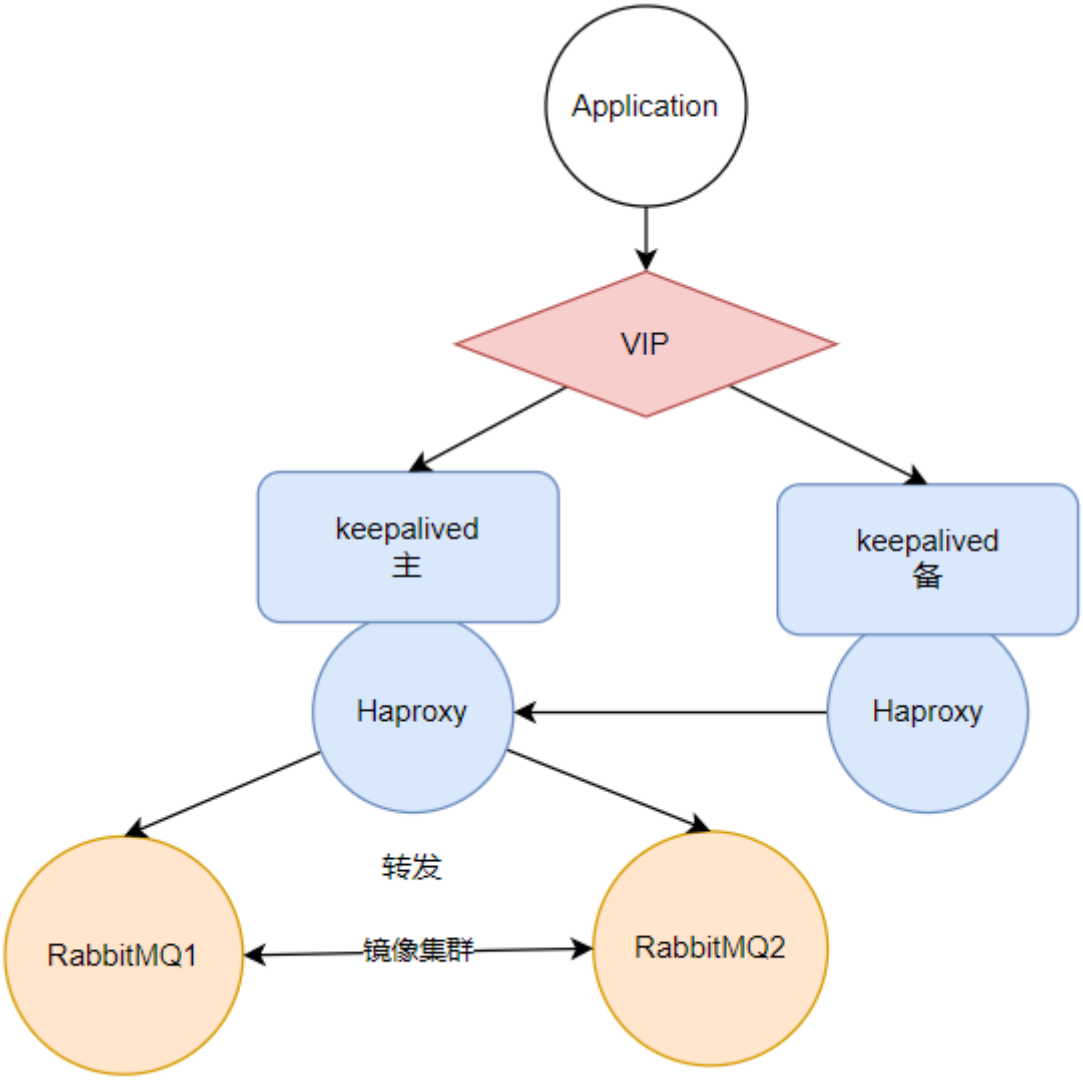

有了haproxy后，如果某一个RabbitMQ服务崩溃了，Haproxy会将请求往另外一个RabbitMQ服务转发，这样应用程序就不需要做IP切换了。此时，对于RabbitMQ来说，整个集群的服务是稳定的。

> Haproxy是一个免费开源的负载均衡工具。类似的工具还有很多，比如F5，nginx等。

2、keepalived防止haproxy单点崩溃

Haproxy保证了RabbitMQ的服务高可用，防止RabbitMQ服务单点崩溃对应用程序的影响。但是同时又带来了Haproxy的单点崩溃问题。如果Haproxy服务崩溃了，整个应用程序就完全无法访问RabbitMQ了。为了防止Haproxy单点崩溃的问题，可以引入keepalived组件来保证Haproxy的高可用。

keepalived是一个搭建高可用服务的常见工具。 他会暴露出一个虚拟IP(VIP)，并将VIP绑定到不同的网卡上。引入keepalived后，可以将VIP先绑定在已有的Haproxy服务上，然后引入一个从Haproxy作为一个备份。 当主Haproxy服务出现异常后，keepalived可以将虚拟IP转为绑定到从Haproxy服务的网卡上，这个过程称为VIP漂移。而对于应用程序，自始至终只需要访问keepalived暴露出来的VIP，感知不到VIP漂移的过程。这样就保证了Haproxy服务的高可用性。



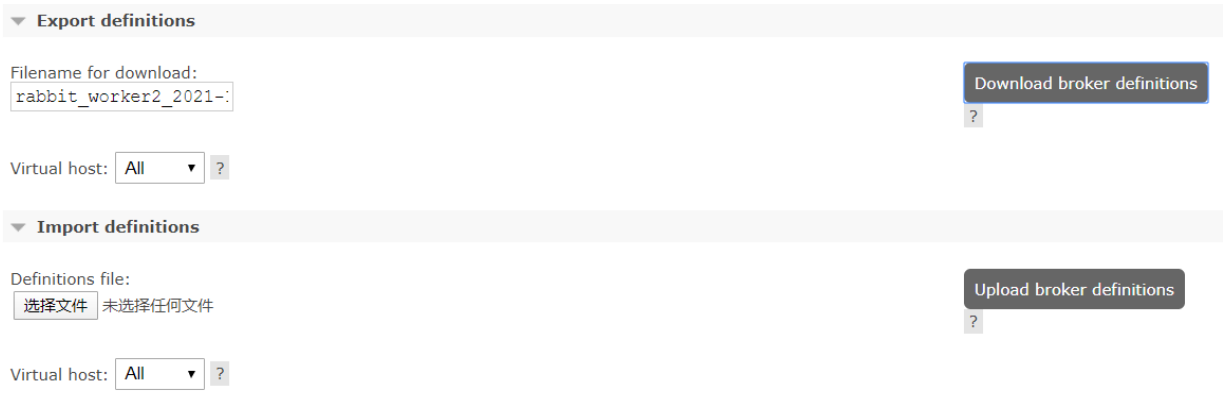

Haproxy+Keeperalived的组合是分布式场景中经常用到的一种高可用方案。他们的部署也不麻烦，就是下载+配置+运行即可。当然，这并不是我们的重点，只要了解即可。如果你对部署操作感兴趣，想要自己搭建一下的话，可以参考下这个文章：https://www.yuque.com/xiaochuan-5hgfq/rqiea6/xc65icrse4kkokeh 官网有对应的搭建视频。

# 5、RabbitMQ的备份与恢复

> 文档地址： https://www.rabbitmq.com/backup.html

RabbitMQ有一个data目录会保存分配到该节点上的所有消息。`我们的实验环境中，默认是在/var/lib/rabbitmq/mnesia目录下` 这个目录里面的备份分为两个部分，一个是元数据(定义结构的数据)，一个是消息存储目录。

**对于元数据，可以在Web管理页面通过json文件直接导出或导入。**




**而对于消息，可以手动进行备份恢复**

其实对于消息，由于MQ的特性，是不建议进行备份恢复的。而RabbitMQ如果要进行数据备份恢复，也非常简单。

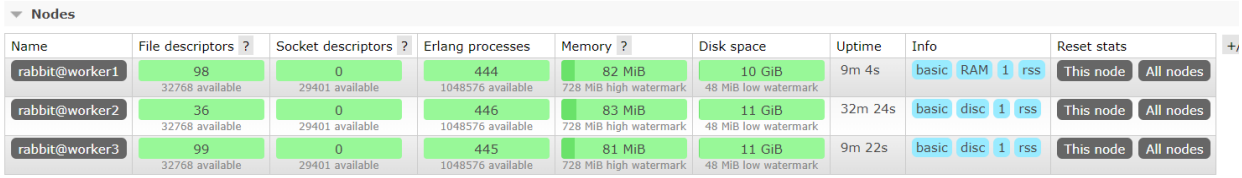
首先，要保证要恢复的RabbitMQ中已经有了全部的元数据，这个可以通过上一步的json文件来恢复。

然后，备份过程必须要先停止应用。如果是针对镜像集群，还需要把整个集群全部停止。

最后，在RabbitMQ的数据目录中，有按virtual hosts组织的文件夹。你只需要按照虚拟主机，将整个文件夹复制到新的服务中即可。持久化消息和非持久化消息都会一起备份。 我们实验环境的默认目录是`/var/lib/rabbitmq/mnesia/rabbit@worker2/msg_stores/vhosts`

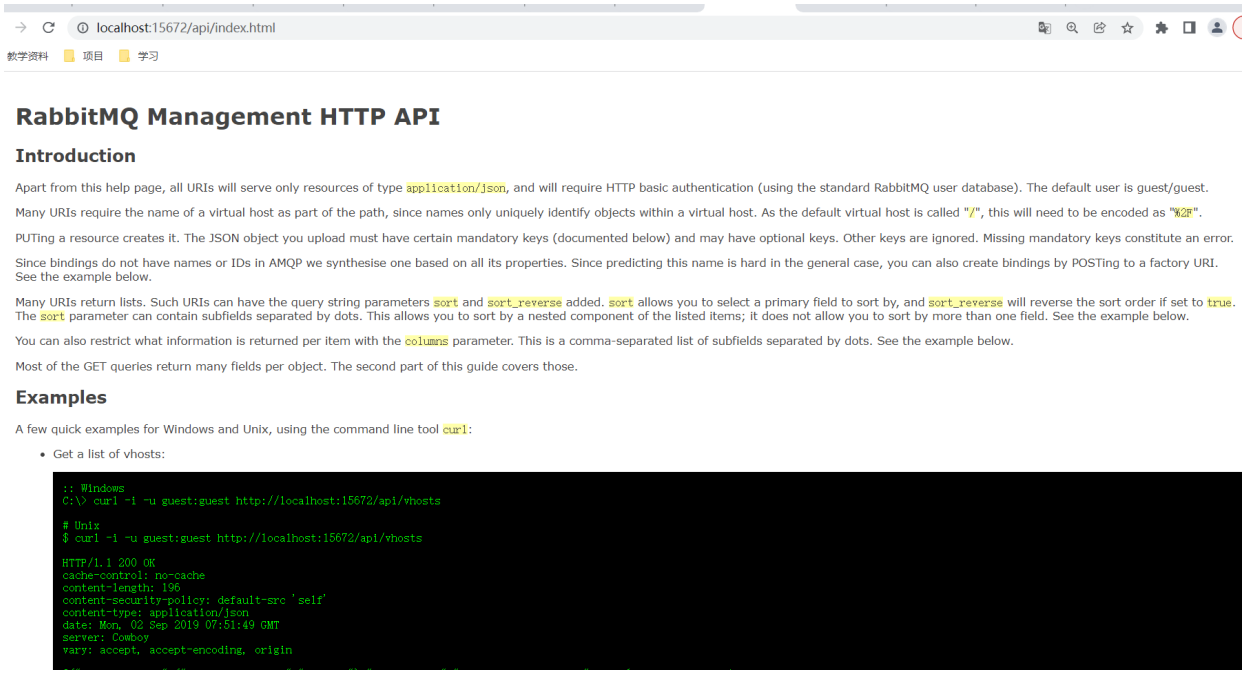
# 6、RabbitMQ的性能监控

关于RabbitMQ的性能监控，在管理控制台中提供了非常丰富的展示。例如在下面这个简单的集群节点图中，就监控了非常多系统的关键资源。




还包括消息的生产消费频率、关键组件使用情况等等非常多的信息，都可以从这个管理控制台上展现出来。但是，对于构建一个自动化的性能监控系统来说，这个管理页面就不太够用了。为此，RabbitMQ也提供了一系列的HTTP接口，通过这些接口可以非常全面的使用并管理RabbitMQ的各种功能。

这些HTTP的接口不需要专门去查手册，在部署的管理控制台页面下方已经集成了详细的文档，我们只需要打开HTTP API的页面就能看到。




比如最常用的 http://[server:port]/api/overview 接口，会列出非常多的信息，包含系统的资源使用情况。通过这个接口，就可以很好的对接Promethus、Grafana等工具，构建更灵活的监控告警体系。

可以看到，这里面的接口相当丰富，不光可以通过GET请求获取各种消息，还可以通过其他类型的HTTP请求来管理RabbitMQ中的各种资源，因此在实际使用时，还需要考虑这些接口的安全性。

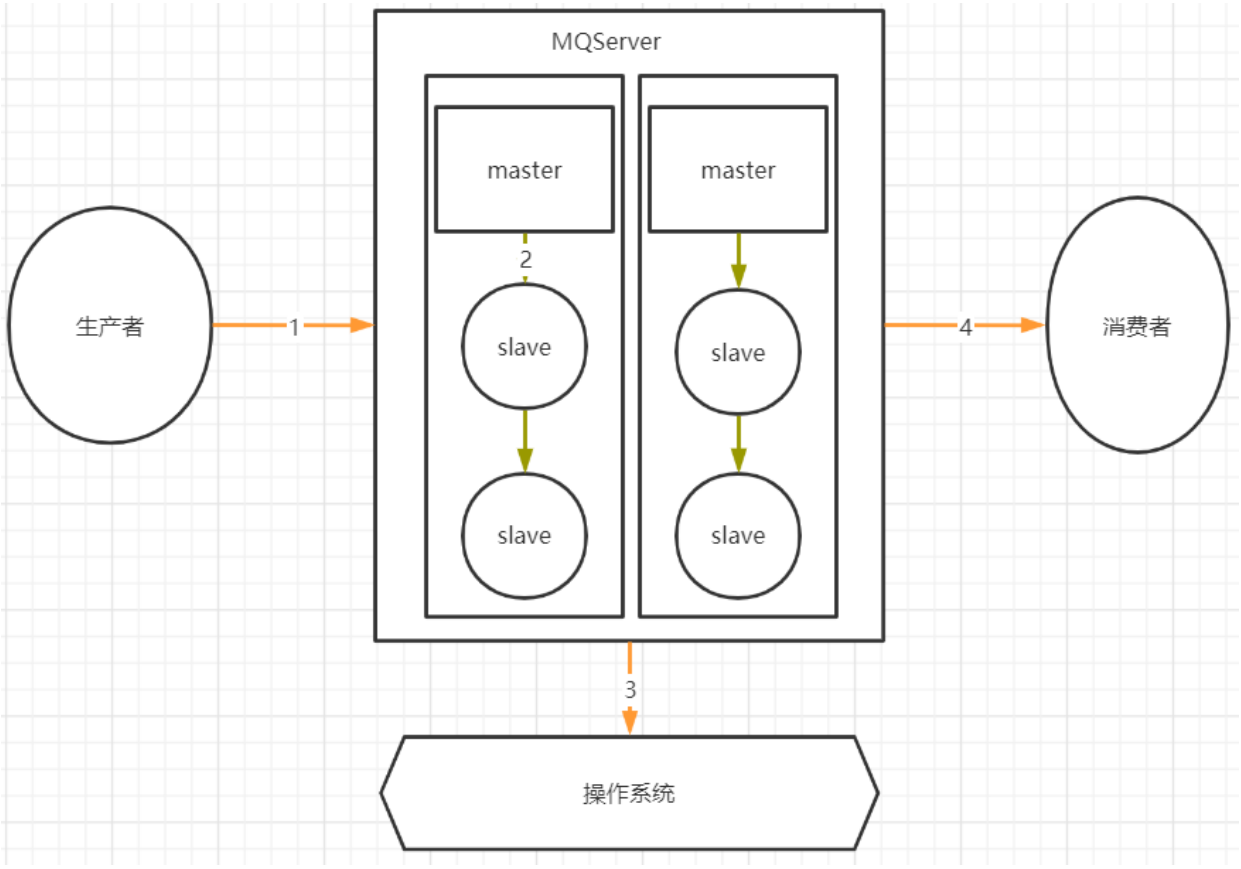
# 二、RabbitMQ如何保证消息不丢失?

这是面试时最喜欢问的问题，其实这是个所有MQ的一个共性的问题，大致的解决思路也是差不多的，但是针对不同的MQ产品会有不同的解决方案。而RabbitMQ设计之处就是针对企业内部系统之间进行调用设计的，所以他的消息可靠性是比较高的。

> 千万不要只回答 手动确认 就结束了。

## 1、哪些环节会有丢消息的可能?

我们考虑一个通用的MQ场景：




其中，1，2，4三个场景都是跨网络的，而跨网络就肯定会有丢消息的可能。

然后关于3这个环节，通常MQ存盘时都会先写入操作系统的缓存page cache中，然后再由操作系统异步的将消息写入硬盘。这个中间有个时间差，就可能会造成消息丢失。如果服务挂了，缓存中还没有来得及写入硬盘的消息就会丢失。这也是任何用户态的应用程序无法避免的。

对于任何MQ产品，都应该从这四个方面来考虑数据的安全性。那我们看看用RabbitMQ时要如何解决这个问题。

# 2、RabbitMQ消息零丢失方案：

## 1》生产者保证消息正确发送到RibbitMQ

对于单个数据，可以使用生产者确认机制。通过多次确认的方式，保证生产者的消息能够正确的发送到RabbitMQ中。

RabbitMQ的生产者确认机制分为同步确认和异步确认。同步确认主要是通过在生产者端使用Channel.waitForConfirmsOrDie()指定一个等待确认的完成时间。异步确认机制则是通过channel.addConfirmListener(ConfirmCallback var1, ConfirmCallback var2)在生产者端注入两个回调确认函数。第一个函数是在生产者消息发送成功时调用，第二个函数则是生产者消息发送失败时调用。两个函数需要通过sequenceNumber自行完成消息的前后对应。sequenceNumber的生成方式需要通过channel的序列获取。int sequenceNumber = channel.getNextPublishSeqNo();

当前版本的RabbitMQ，可以在Producer中添加一个ReturnListener，监听那些成功发到Exchange，但是却没有路由到Queue的消息。如果不想将这些消息返回给Producer，就可以在Exchange中，也可以声明一个alternate-exchange参数，将这些无法正常路由的消息转发到指定的备份Exchange上。

如果发送批量消息，在RabbitMQ中，另外还有一种手动事务的方式，可以保证消息正确发送。

手动事务机制主要有几个关键的方法： channel.txSelect() 开启事务； channel.txCommit() 提交事务；channel.txRollback() 回滚事务； 用这几个方法来进行事务管理。但是这种方式需要手动控制事务逻辑，并且手动事务会对channel产生阻塞，造成吞吐量下降

## 2》 RabbitMQ消息存盘不丢消息

这个在RabbitMQ中比较好处理，对于Classic经典队列，直接将队列声明成为持久化队列即可。而新增的Quorum队列和Stream队列，都是明显的持久化队列，能更好的保证服务端消息不会丢失。

## 3》 RabbitMQ 主从消息同步时不丢消息

这涉及到RabbitMQ的集群架构。首先他的普通集群模式，消息是分散存储的，不会主动进行消息同步了，是有可能丢失消息的。而镜像模式集群，数据会主动在集群各个节点当中同步，这时丢失消息的概率不会太高。

另外，启用Federation联邦机制，给包含重要消息的队列建立一个远端备份，也是一个不错的选择。

## 4》 RabbitMQ消费者不丢失消息

RabbitMQ在消费消息时可以指定是自动应答，还是手动应答。如果是自动应答模式，消费者会在完成业务处理后自动进行应答，而如果消费者的业务逻辑抛出异常，RabbitMQ会将消息进行重试，这样是不会丢失消息的，但是有可能会造成消息一直重复消费。

将RabbitMQ的应答模式设定为手动应答可以提高消息消费的可靠性。

```
channel.basicConsume(queueName, false, new DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
                                       BasicProperties properties, byte[] body)
                    throws IOException {
                long deliveryTag = envelope.getDeliveryTag();
                channel.basicAck(deliveryTag, false);
            }
        });
channel.basicConsume(queueName, true, myconsumer);
```

另外这个应答模式在SpringBoot集成案例中，也可以在配置文件中通过属性 spring.rabbitmq.listener.simple.acknowledge-mode 进行指定。可以设定为 AUTO 自动应答； MANUAL 手动应答；NONE 不应答； 其中这个NONE不应答，就是不启动应答机制，RabbitMQ只管往消费者推送消息后，就不再重复推送消息了，相当于RocketMQ的sendoneway， 这样效率更高，但是显然会有丢消息的可能。

最后，任何用户态的应用程序都无法保证绝对的数据安全，所以，备份与恢复的方案也需要考虑到。

# 三、如何保证消息幂等?

1、RabbitMQ的自动重试功能：

当消费者消费消息处理业务逻辑时，如果抛出异常，或者不向RabbitMQ返回响应，默认情况下，RabbitMQ会无限次数的重复进行消息消费。

处理幂等问题，**首先要设定RabbitMQ的重试次数。**在SpringBoot集成RabbitMQ时，可以在配置文件中指定spring.rabbitmq.listener.simple.retry开头的一系列属性，来制定重试策略。

**然后，需要在业务上处理幂等问题。**

处理幂等问题的关键是要给每个消息一个唯一的标识。

在SpringBoot框架集成RabbitMQ后，可以给每个消息指定一个全局唯一的MessageID，在消费者端针对MessageID做幂等性判断。关键代码：

```
//发送者指定ID字段
Message message2 =
MessageBuilder.withBody(message.getBytes()).setMessageId(UUID.randomUUID().toString(
)).build();
        rabbitTemplate.send(message2);
//消费者获取MessageID，自己做幂等性判断
@RabbitListener(queues = "fanout_email_queue")
public void process(Message message) throws Exception {
    // 获取消息Id
    String messageId = message.getMessageProperties().getMessageId();
    ...
}
```

> 要注意下这里用的message要是org.springframework.amqp.core.Message

在原生API当中，也是支持MessageId的。当然，在实际工作中，最好还是能够添加一个具有业务意义的数据作为唯一键会更好，这样能更好的防止重复消费问题对业务的影响。比如，针对订单消息，那就用订单ID来做唯一键。在RabbitMQ中，消息的头部就是一个很好的携带数据的地方。

```
// ==== 发送消息时，携带sequenceNumber和orderNo
AMQP.BasicProperties.Builder builder = new AMQP.BasicProperties.Builder();
builder.deliveryMode(MessageProperties.PERSISTENT_TEXT_PLAIN.getDeliveryMode());
builder.priority(MessageProperties.PERSISTENT_TEXT_PLAIN.getPriority());
//携带消息ID
builder.messageId(""+channel.getNextPublishSeqNo());
Map<String, Object> headers = new HashMap<>();
//携带订单号
headers.put("order", "123");
builder.headers(headers);
channel.basicPublish("", QUEUE_NAME, builder.build(), message.getBytes("UTF-8"));

// ==== 接收消息时，拿到sequenceNumber
Consumer myconsumer = new DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
                    BasicProperties properties, byte[] body)
                    throws IOException {
                //获取消息ID
                System.out.println("messageId:"+properties.getMessageId());
                //获取订单ID
                 properties.getHeaders().forEach((key,value)->
System.out.println("key: "+key +"; value: "+value));
```

```
                // (process the message components here ...)
                //消息处理完后，进行答复。答复过的消息，服务器就不会再次转发。
                //没有答复过的消息，服务器会一直不停转发。
                channel.basicAck(deliveryTag, false);
            }
        };
channel.basicConsume(QUEUE_NAME, false, myconsumer);
```

# 四、如何保证消息的顺序?

某些场景下，需要保证消息的消费顺序，例如一个下单过程，需要先完成扣款，然后扣减库存，然后通知快递发货，这个顺序不能乱。如果每个步骤都通过消息进行异步通知的话，这一组消息就必须保证他们的消费顺序是一致的。

在RabbitMQ当中，针对消息顺序的设计其实是比较弱的。唯一比较好的策略就是 单队列+单消息推送。即一组有序消息，只发到一个队列中，利用队列的FIFO特性保证消息在队列内顺序不会乱。但是，显然，这是以极度消耗性能作为代价的，在实际适应过程中，应该尽量避免这种场景。

然后在消费者进行消费时，保证只有一个消费者，同时指定prefetch属性为1，即每次RabbitMQ都只往客户端推送一个消息。像这样：

```
spring.rabbitmq.listener.simple.prefetch=1
```

而在多队列情况下，如何保证消息的顺序性，目前使用RabbitMQ的话，还没有比较好的解决方案。在使用时，应该尽量避免这种情况。

# 五、关于RabbitMQ的数据堆积问题

RabbitMQ一直以来都有一个缺点，就是对于消息堆积问题的处理不好。当RabbitMQ中有大量消息堆积时，整体性能会严重下降。而目前新推出的Quorum队列以及Stream队列，目的就在于解决这个核心问题。但是这两种队列的稳定性和周边生态都还不够完善，目前大部分企业还是围绕Classic经典队列构建应用。因此，在使用RabbitMQ时，还是要非常注意消息堆积的问题。尽量让消息的消费速度和生产速度保持一致。

而如果确实出现了消息堆积比较严重的场景，就需要从数据流转的各个环节综合考虑，设计适合的解决方案。

首先在消息生产者端：

对于生产者端，最明显的方式自然是降低消息生产的速度。但是，生产者端产生消息的速度通常是跟业务息息相关的，一般情况下不太好直接优化。但是可以选择尽量多采用批量消息的方式，降低IO频率。

然后在RabbitMQ服务端：

从前面的分享中也能看出，RabbitMQ本身其实也在着力于提高服务端的消息堆积能力。对于消息堆积严重的队列，可以预先添加懒加载机制，或者创建Sharding分片队列，这些措施都有助于优化服务端的消息堆积能力。另外，尝试使用Stream队列，也能很好的提高服务端的消息堆积能力。

接下来在消息消费者端：

要提升消费速度最直接的方式，就是增加消费者数量了。尤其当消费端的服务出现问题，已经有大量消息堆积时。这时，可以尽量多的申请机器，部署消费端应用，争取在最短的时间内消费掉积压的消息。但是这种方式需要注意对其他组件的性能压力。

对于单个消费者端，可以通过配置提升消费者端的吞吐量。例如

```
# 单次推送消息数量
spring.rabbitmq.listener.simple.prefetch=1
# 消费者的消费线程数量
spring.rabbitmq.listener.simple.concurrency=5
```

灵活配置这几个参数，能够在一定程度上调整每个消费者实例的吞吐量，减少消息堆积数量。

当确实遇到紧急状况，来不及调整消费者端时，可以紧急上线一个消费者组，专门用来将消息快速转录。保存到数据库或者Redis，然后再慢慢进行处理。

# 六、课程总结

基于MQ的事件驱动机制，给庞大的互联网应用带来了不一样的方向。MQ的异步、解耦、削峰三大功能特点在很多业务场景下都能带来极大的性能提升，在日常工作过程中，应该尝试总结这些设计的思想。

虽然MQ的功能，说起来比较简单，但是随着MQ的应用逐渐深化，所需要解决的问题也更深入。对各种细化问题的挖掘程度，很大程度上决定了开发团队能不能真正Hold得住MQ产品。通常面向互联网的应用场景，更加注重MQ的吞吐量，需要将消息尽快的保存下来，再供后端慢慢消费。而针对企业内部的应用场景，更加注重MQ的数据安全性，在复杂多变的业务场景下，每一个消息都需要有更加严格的安全保障。而在当今互联网，Kafka是第一个场景的不二代表，但是他会丢失消息的特性，让kafka的使用场景比较局限。RabbitMQ作为一个老牌产品，是第二个场景最有力的代表。当然，随着互联网应用不端成熟，也不断有其他更全能的产品冒出来，比如阿里的RocketMQ以及雅虎的Pulsar。但是不管未来MQ领域会是什么样子，RabbitMQ依然是目前企业级最为经典也最为重要的一个产品。他的功能最为全面，周边生态也非常成熟，并且RabbitMQ有庞大的Spring社区支持，本身也在吸收其他产品的各种优点，持续进化，所以未来RabbitMQ的重要性也会更加凸显。

最后，整个课程内容其实是比较多的。同时为了让这些内容能够尽量让你接触到，所以整理出了大量的资料、配置、试验。希望能够带你深入理解RabbitMQ解决各种常见问题的思路。你可能对RabbitMQ这么个年代久远的产品有过了解，但是，课上这些资料、试验你还是一定要自己从头梳理一下。因为RabbitMQ这个产品，沉淀了这么多年，积累起来的业务经验实在是太多为了。你就算用过很多年RabbitMQ，也几乎不可能触摸到RabbitMQ的全部。并且在这个过程中，你一定能找到一些你平时没有注意的技术细节。这些细节或许就是你日后解决某一个实际问题的关键。

> 有道云笔记链接：https://note.youdao.com/s/Lno1d7P0