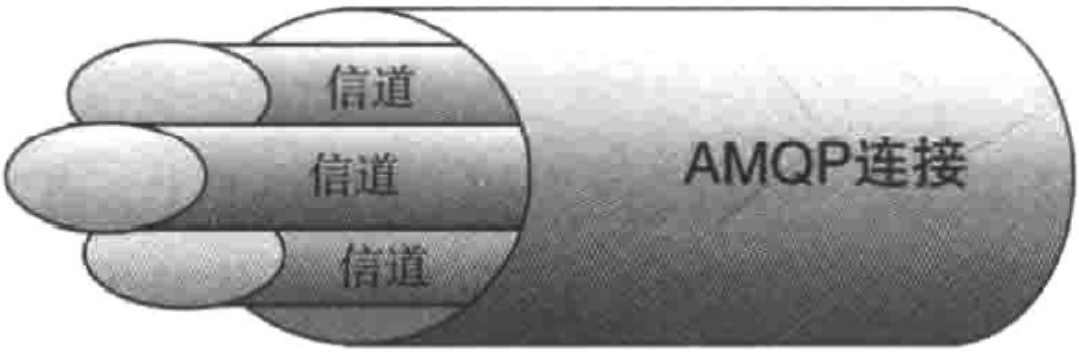
RabbitMQ实战 高效部署分布式消息队列

第2章 理解消息通信

2.2 此底部开始构造：队列

信道(channel)是建立在TCP连接上的虚拟连接。因为如果每个生产者/消费者都使用真实的TCP连接的话，每一个线程连接到RabbitMQ，都会建立一个TCP连接，不仅会造成TCP连接的巨大浪费(创建和销毁TCP回话开销很大)，而且操作系统每秒也就只能建立这点数量的连接，很快就会碰到性能瓶颈。



如图，一条电缆(TCP连接)有许多光纤束(信道)，运行所有连接的线程通过多条光纤束同时进行传输和接收。

当一个RabbitMQ队列有多个消费者时，队列收到的消息将以循环的方式发送给消费者，每个消息只发送给一个订阅的消费者。

- 消费者接收到的每一条消息都必须进行确认
- 或者在订阅到队列的时候讲auto_ack参数设置为true

当设置了auto_ack时，一旦消费者接收消息，RabbitMQ会自动视其确认了消息。消费者通过确认命令告诉RabbitMQ它已经正确接收了消息，此时RabbitMQ才能安全地把消息从队列中删除。

如果消费者收到一条消息，但是确认之前从Rabbit断开连接(或者从队列上取消了订阅)，RabbitMQ会认为这条消息没有分发，**然后重新分发给下一个订阅的消费者。**另一方面，如果消费者没有确认消息，**Rabbit讲不会给该消费者发送更多消息**，这是因为在上一条消息被确认之前，Rabbit会认为这个消费者没有准备好接收下一条消息。

2.3 联合起来：交换器和绑定

队列通过路由键(routing key)绑定到交换器

四种类型交换器：

- direct

  服务器包含一个空白字符串名称的默认交换器，当声明一个队列时，它会自动绑定到默认交换器，并以队 列名称作为路由键。

- fanout

这种类型的交换器会将收到的消息广播到绑定的队列上。例如一个web应用需要在用户上传新图片时，用 户相册必须清除缓存，同时用户应该得到积分奖励。这时可以将两个队列绑定到图片上传交换器上，一个 用于清除缓存，另一个用于增加用户积分。如果产品需要增加新功能时，就可以不修改原来代码，只需要 新声明一个队列并绑定到fanout交换器上。

- topic

  可以使用通配符(*和#)

- headers

headers交换器和direct完全一致，但性能会差很多，因此并不实用。

2.4 多租户模式：虚拟主机和隔离

每一个RabbitMQ服务器都能创建虚拟消息服务器，我们称之为虚拟主机(vhost)，每个vhost本质上是一个mini版的RabbitMQ服务器，拥有自己的队列，交换器和绑定，**已经自己的权限机制。**vhost之于Rabbit就像虚拟机之于物理服务器一样：它既能将同一个Rabbit的众多客户区分开来，又可以避免队列和交换器的命名冲突。

当你在Rabbit里创建一个用户时，用户通常会被指派给至少一个vhost，并且只能访问被指派的vhost内的队列、交换器和绑定。

当在RabbitMQ集群上创建vhost时，整个集群上都会创建该vhost。

2.5 我的消息去哪了呢？持久化和你的策略

默认情况下，重启RabbitMQ服务器后，那些队列(连同里面的消息)和交换器都会消失，原因在于每个队列和交换器的durable属性(默认为false)，将它设置为true，在崩溃或重启后就不需要重新创建队列(或者交换器)。***但是这样不能保证消息幸免于重启。***

如果消息要从Rabbit崩溃中恢复，那么消息必须：

- 把它的"投递模式"选项设置为2来把消息标记成持久化
- 它还必须被发布到持久化的交换器中
- 到达持久化的队列中

当发布一条持久性消息到持久交换器上时，Rabbit会在消息提交到持久化日志文件后才提交响应。***之后这条消息如果路由到了非持久队列中，它会自动从持久性日志中移除，并且无法从服务器重启中恢复。***一旦从持久化队列中消费了一条持久性消息的话(并且确认了它)。RabbitMQ会在持久化日志中把这条消息标记为等待垃圾收集。在消费持久性消息前如果RabbitMQ重启，服务器会自动重建交换器和队列(已经绑定)，重播持久性日志文件中的消息到合适的队列或者交换器上(取决于Rabbit服务器宕机的时候，消息处在路由过程的哪个环节)。

可以通过其他方式保证消息投递，例如：生产者可以在单独的信道上监听应答队列，每次发送消息的时候，都包含应答队列的名称，这样消费者就可以回发应答到该队列以确认接收到了，如果消息应答未在合理时间内到达，生产者就重发消息。

由于发布操作不返回任何信息给生产者，生产者怎么知道服务器是否已经持久化了持久消息到硬盘呢？服务器可能会在把消息写入硬盘前就宕机了，消息因此而丢失，这就是***事务***发挥作用的地方。在AMQP中，把信道设置成事务模式，就可以通过信道发送那些想要确认的消息，之后还有多个其他AMQP命令，如果事务中首次发布成功，信道会在事务中完成其他AMQP命令，如果发送失败，其他AMQP命令将不会执行。

> 事务会降低大约2~10倍的消息吞吐量，而且会使生产者应用程序产生同步。

RabbitMQ有更好的方案来保证消息投递：***发送方确认模式***。告诉Rabbit将信道设置为confirm模式，而且只能通过重新创建信道来关闭该设置。一旦信道进入confirm模式，所有在信道上发布的消息都会被指派一个唯一的ID号。一旦消息被投递给所有匹配的队列后，信道会发送一个发送方确认模式给生产者(包含消息的唯一ID)，这使得生产者知晓消息已经安全到达目的队列。如果消息和队列时可持久化的，那么确认消息只会在队列将消息写入磁盘后才会发出，

> 发送发确认模式的最大好处是异步，一旦发布一条消息，生产者就可以在等待确认的同时继续发送下一条，当确认消息收到时，生产者应用的回调方法就会被触发来处理该确认消息，如果Rabbit发生了内部错误导致消息丢失，Rabbit会发送一条nack(未确认)消息，就像发送确认消息那样，只不过这次说明的是消息已经丢失。由于没有消息回滚(和事务相比)，发送法确认模式更加轻量级。

2.6 把所有内容结合起来：一条消息的一生

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

```
@Configuration
public class RabbitMQConfig {

    @Bean
    public Queue helloQueue() {
        return new Queue("hello_queue");
    }

    @Bean
    public Queue userQueue() {
        return new Queue("user_queue");
    }

    @Bean
    public Queue queueMessage() {
        return new Queue("topicMessage_queue");
    }

    @Bean
    public Queue queueMessages() {
        return new Queue("topicMessages_queue");
    }

    @Bean
    public Queue AMessage() {
        return new Queue("fanoutA_queue");
    }

    @Bean
    public Queue BMessage() {
        return new Queue("fanoutB_queue");
    }
```

```
    @Bean
    public Queue CMessage() {
        return new Queue("fanoutC_queue");
    }

    @Bean
    DirectExchange directExchange() {
        return new DirectExchange("directExchange");
    }

    @Bean
    TopicExchange topicExchange() {
        return new TopicExchange("topicExchange");
    }
    @Bean
    FanoutExchange fanoutExchange() {
        return new FanoutExchange("fanoutExchange");
    }

    /**
     * 将userQueue()和directExchange()通过"user"绑定
     * @return
     */
    @Bean
    Binding bindingDirectExchange() {
        return BindingBuilder.bind(userQueue()).to(directExchange()).with("user");
    }

    /**
     * 将队列topic.message与exchange绑定，binding_key为topic.message,就是完全匹配
     * @return
     */
    @Bean
    Binding bindingExchangeMessage() {
        return BindingBuilder.bind(queueMessage()).to(topicExchange()).with("topic.message");
    }

    /**
     * 将队列topic.messages与exchange绑定，binding_key为topic.#,模糊匹配
     * @return
     */
    @Bean
    Binding bindingExchangeMessages() {
        return BindingBuilder.bind(queueMessages()).to(topicExchange()).with("topic.#");
    }

    @Bean
    Binding bindingExchangeA() {
        return BindingBuilder.bind(AMessage()).to(fanoutExchange());
    }

    @Bean

    Binding bindingExchangeB() {
```

```
        return BindingBuilder.bind(BMessage()).to(fanoutExchange());
    }

    @Bean
    Binding bindingExchangeC() {
        return BindingBuilder.bind(CMessage()).to(fanoutExchange());
    }
}
```

```
/**
 * 生产者
 */
@Component
public class HelloSender1 {
    private final static Logger LOGGER = LogManager.getLogger(HelloSender1.class);

    @Autowired
    private AmqpTemplate rabbitTemplate;

    /**
     * direct类型的交换器：通过默认的exchange，routing_key就是queueName：“hello_queue”
     */
    public void send() {
        String sendMsg = "hello1 " + new Date();
        LOGGER.info("Sender1 : {}", sendMsg);
        this.rabbitTemplate.convertAndSend("hello_queue", sendMsg);
    }

    /**
     * direct类型的交换器：通过directExchange，routing_key就是"user"
     * @param age
     */
    public void send(int age) {
        User user = new User();
        user.setAge(age);
        user.setName("张三");
        user.setGender("male");
        user.setBirthday(new Date());
        LOGGER.info("Sender1 : {}", user);
        this.rabbitTemplate.convertAndSend("directExchange", "user", user);
    }
}
```

```
/**
 * 消费者
 */
@Component
public class HelloReceiver1 {
    private final static Logger LOGGER = LogManager.getLogger(HelloReceiver1.class);


    @RabbitHandler
```

```
    @RabbitListener(queues = "hello_queue")
    public void process(String text) {
        LOGGER.info("Receiver2 : {}", text);
    }

    @RabbitHandler
    @RabbitListener(queues = "user_queue")
    public void process(User user) {
        LOGGER.info("Receiver2 : {}", user);
    }
}
```

第3章 运行和管理Rabbit

3.1 服务器管理

3.1.1 启动节点

Erlang也有虚拟机，虚拟机的每个实例我们称为节点。多个Erlang程序可以运行在同一个节点上。节点间可以进行本地通信(不管它们是否真的在同一台服务器上)。如果程序崩溃了(例如RabbitMQ崩溃了)，Erlang节点会自动尝试重启应用程序(前提是Erlang没有崩溃)。

RabbitMQ使得启动Erlang节点和Rabbit应用很简单，只需要在安装目录下找到`./sbin`目录，运行`./rabbitmq-server`。如果在启动过程中遇到错误，检查一下日志，通常情况下在`/var/log/rabbitmq/`目录下找到名为`rabbit@[hostname].log`的日志文件，也可以通过`./rabbitmq-server -detached`以守护程序的方式在后台运行。

3.1.2 停止节点

运行`./rabbitmqctl stop`时，rabbitmqctl会和本地节点通信并指示其干净地关闭，可以指定关闭不同节点，只需传入`-n rabbit@[hostname]`

3.1.3 关闭和重启应用程序：有何差别

3.1.2节中的命令会关闭RabbitMQ节点(应用程序和Erlang节点)。

只停止RabbitMQ，只需运行`./rabbitmqctl stop_app`

3.1.4 Rabbit配置文件

配置文件在`/etc/rabbitmq/rabbitmq.config`

配置文件的格式

```
[
    {mnesia, [{dump_log_write_threshold, 1000}]},
    {rabbit, [{vm_memory_high_watermark, 0.4}]}
].
```

mnesia指的是Mnesia数据库配置选项，rabbit指的是RabbitMQ特定的配置选项。

> RabbitMQ中的每个队列、交换器和绑定的元数据(除了消息的内容)都是保存在Mnesia的，Mnesia通过将RabbitMQ元数据首先写入一个仅限追加的日志文件，再定期将日志内容转储到真实的Mnesia数据库文件中。Mnesia的dump_log_write_threshold 控制转储的频度，1000就是每1000个条目就转储日志内容到数据库文件。设置更高的数值将减少I/O负载并增加持久化消息的性能

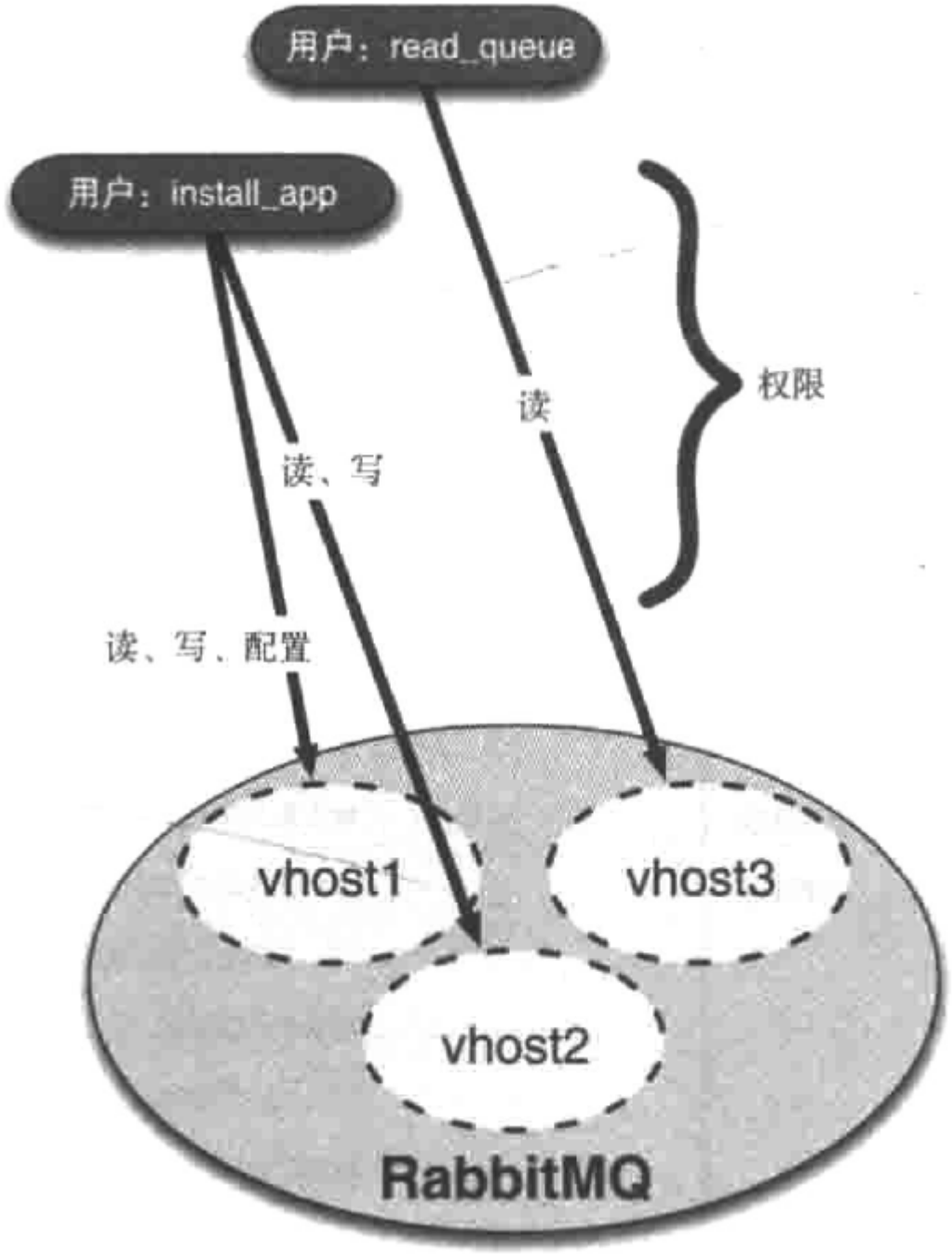
部分Rabbit配置选项：

选项名称	值类型	默认值	描述
vm_memory_high_watermark	十进制百分数	0.4	RabbitMQ运行小号的内存(0.4=40%)
msg_store_file_size_limit	整型(字节)	16777216	RabbitMQ垃圾手机存储内容之前，消息存储数据库的最大大小
queue_index_max_journal_entries	整型	262144	在转储到消息存储数据库并提交之前，消息存储日志里的最大条目数

配置文件无法修改RabbitMQ的访问控制

3.2 请求许可

首先创建用户，然后为其赋予权限



单个用户可以跨越多个bhost进行授权

3.2.1 管理用户

在RabbitMQ中，用户是访问控制的基本单元。

创建用户通过`./rabbitmqctl add_user hermionc 123456`，用户名是hermionc，密码是123456，删除用户`./rabbitmqctl delete_user hermionc`。删除用户时，***任何引用该用户的访问控制条目都会从Rabbit权限数据库中自动删除。***

查看用户：`./rabbitmqctl list_users`。

更改密码：`./rabbitmqctl change_password hermionc 12345678`

3.2.2 Rabbit的权限控制

权限：

- 读

  有关消费信息的任何操作，包括"清除"整个队列

- 写

  发布消息

- 配置

  队列和交换器的创建和删除

如果有名为sycamore的vhost，想要授予hermionc完全的访问权限(配置、写和读)，需要执行：`./rabbitmqctl set_permissions -p sycamore hermionc ".*" ".*" ".*"`

- `-p sycamore`：告诉set_permissions条目应该应用到哪个vhost上，这里是用到sycamore这个vhost上。
- `hermionc`：被授予权限的用户
- `".*" ".*" ".*"`：这是授予的权限，这些值分别映射到配置、写和读

三个权限值中的每一个都是正则表达式。

例如如果想为hermionc用户授予在oak vhost上的权限，运行该用户所有读操作，只能对以checks-开始的队列和交换器执行写操作，阻止配置操作：`./rabbitmqctl set_permissions -p oak -s all hermionc "" "checks-.*" ".*"`

移除hermionc在任何vhost上(例如oak vhost)的权限：`./rabbitmqctl clear_permissions -p oak hermionc`

查看所有用户的权限：`./rabbitmqctl list_permissions -p oak`

查看用户在RabbitMQ服务器所有vhost上的权限：`./rabbitmqctl list_user_permissions hermionc`

3.3 检查

3.3.1 查看数据统计

查看队列：`./rabbitmqctl list_queues`，加上`-p sycamore`选项可以展示在sycamore vhost上的队列。

查看交换器：`./rabbitmqctl list_exchanges`

3.3.2 理解RabbitMQ日志

可以使用AMQP交换器、队列和绑定来实时获得日志，并对此做出反应

在rabbitmq-server脚本中：

```
LOG_BASE=/var/log/rabbitmq
```

在这个文件夹内RabbitMQ会创建两个日志文件：RABBITMQ_NODENAME-sasl.log和RABBITMQ_NODENAME.log。RABBITMQ_NODENAME指的是`_rabbit@localhost_`或者就是`rabbit`，这取决于如何配置系统。

当RabbitMQ记录Erlang相关信息时，将日志写入sasl.log中。如果想看服务器正在发生的事件，可以：`tail -f rabbit.log`

RabbitMQ中有一个叫做***amq.rabbitmq.log***的topic交换器，RabbitMQ把日志信息发布到该交换器上，并以严重级别作为路由键：error、warning和info。可以创建一个消费者监听日志并做出响应的反应。

3.4 修复Rabbit：疑难解答

由badrpc、nodedown和其他Erlang引起的问题

像badrpc，nodedown这样的消息是由Erlang虚拟机产生的。这些错误经常在尝试使用`rabbitmqctl`命令的时候发生

rabbitmqctl会启动Erlang节点，并从那里使用Erlang分布式系统尝试连接RabbitMQ节点，这需要两样东西：***合适的Erlang cookie*** 和 ***合适的节点名称***。Erlang节点通过交换作为秘密令牌的Erlang cookie以获得认证。Erlang讲令牌存储在：`~/.erlang.cookie`中。

Mnesia和主机名

RabbitMQ启动时做的第一件事就是启动Mnesia数据库，导致Mnesia启动失败的原因大致有两个：

- MNESIA_BASE目录的权限问题，运行RabbitMQ服务器的用户需要对该文件夹的写权限
- Mnesia读取表格失败，如果主机名更改了，或是服务器运行在集群模式下，无法在启动时候连接到其他节点，这些都会导致失败。

第4章 解决Rabbit相关问题：编码与模式

4.1 解耦风雨路：谁将我们推向消息通信

4.1.2 提供扩展性：没有负载均衡器的世界

RabbitMQ会将消息在多个消费者之间平级地分发，相当于一个免费的负载均衡

第5章 集群并处理失败

5.1 开足马力：RabbitMQ集群

RabbitMQ最优秀的功能之一就是其***内建集群***

当一个Rabbit集群节点崩溃时，该节点上队列的消息也会消失，这是因为RabbitMQ默认不会将队列的内容复制懂啊整个集群上。如果不进行特别的配置，这些消息仅存在于队列所属的那个节点上。

5.2 集群架构

RabbitMQ会始终记录以下四种类型的内部元数据：

- 队列元数据：队列名称和它们的属性（是否可持久化，是否自动删除）
- 交换器元数据：交换器名称、类型和属性（可持久化）
- 绑定元数据：一张简单的表格展示了如何将消息路由到队列
- vhost元数据：为vhost内的队列、交换器和绑定提供命名空间和安全属性

在一个节点中，RabbitMQ会将这些信息存储在内存中，同时将那些标记为可持久化的队列和交换器（已经它们的绑定）存储到硬盘上。当引入集群时，RabbitMQ需要追踪新的元数据类型：集群节点位置，以及节点与已记录的其他类型元数据的关系。集群给其中的单个节点提供了选择：将元数据存储到RAM和磁盘中（单节点集群的默认设置），或者仅存储在RAM中。

5.2.1 集群中的队列

将两个节点组成集群时，***不是每一个节点都有所有队列的完全拷贝***。在单一节点设置中，所有关于队列的信息都完全存储子该节点上。但是如果在集群中创建队列的话，***集群只会在单个节点而不是在所有节点上创建完整的队列信息***，结果只有队列的所有者节点知道有关队列的所有信息。所有其他非所有者节点只知道队列的元数据和指向该队列存在的那个节点的指针。

所以当集群中的一个节点挂掉，这个节点上的队列和绑定都会消失，监听这个队列的消费者就会丢失订阅，并且任何匹配该队列绑定信息的新信息也会丢失。

如果队列最开始被设置成了可持久化，这时让消费者重连到集群来重新创建队列的话会返回一个`404 NOT_FOUND`，这是因为队列被标识为持久化的，如果允许消费者重建队列的话，宕机节点上的持久化队列信息就会丢失。***此时想要该队列重回集群的唯一方法就只有恢复宕机节点***。但是如果消费者尝试重建的队列不是可持久化的，那么重新声明就会成功。

为什么默认情况下RabbitMQ不将队列内容和状态复制到所有节点上呢？

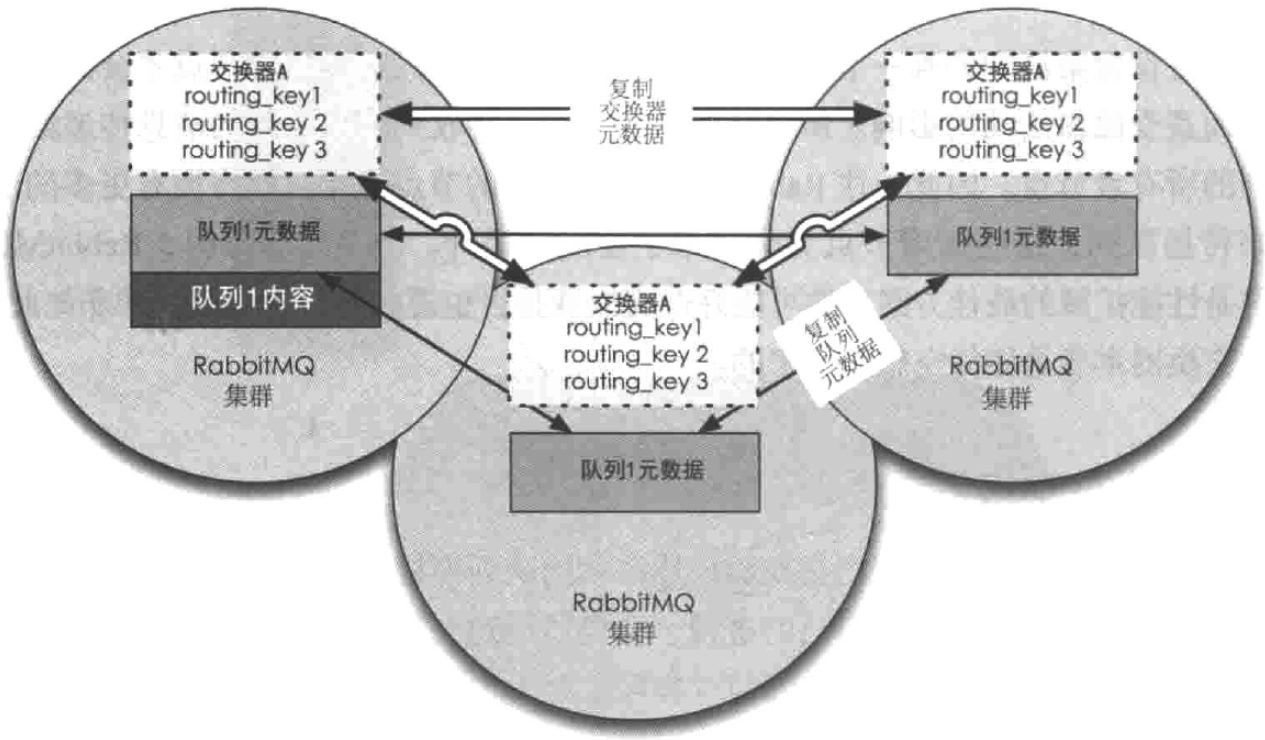
- 存储空间，如果每个集群节点都拥有所有队列的完整拷贝，那么添加新的节点不会带来更多存储空间。
- 性能，消息的发布需要将消息复制到每个集群节点，对于持久化消息来说，每一条消息都会触发磁盘活动。

听歌设置集群中的唯一节点来负责任何特定队列，只有该负责节点才会因队列消息而遭受磁盘活动的影响，所有其他节点需要将接收到的该队列的消息传递给该队列的所有者节点。

5.2.2 分布交换器

交换器本质上只是一个名称和一个队列绑定列表，***信道才是真正的路由器***。所有交换器不会像集群中的队列那样收到限制。

由于交换器本质上是一张查询表，因此将交换器在整个集群中进行复制会更加简单。***集群中的每个节点拥有每个交换器的所有信息***。



当消息已经被发布到信道上，但在路由完成之前节点发送故障，此时可通过AMQP事务或者发送法确认模式来确保应用程序一直发布而不丢失一条消息。

5.2.3 是内存节点还是磁盘节点

每个RabbitMQ节点，不管是单一节点系统还是集群中的一部分，要么是内存节点，要么是磁盘节点。

- 内存节点将所有队列、交换器、绑定、用户、权限和vhost的元数据定义都仅存储在内存中
- 磁盘节点则将元数据既存在内存中也存储在磁盘中

> 单节点集群只允许磁盘类型的节点，否则每次重启RabbitMQ后，所有关于系统的配置信息都会丢失。

在集群中，可以配置部分节点为内存节点。为什么选择将元数据仅存储在内存中？***因为它使得像队列和交换器声明之类的操作更加快速***。

当在集群中声明队列、交换器和绑定的时候，这些操作会直到所有集群节点都成功提交元数据变更后才返回，对于内存节点，这意味着将变更写入内存，而对于磁盘节点，这意味昂贵的磁盘写入操作。

RabbitMQ只要求在集群中至少有一个磁盘节点，所有其他节点都可以是内存节点。当节点加入或者离开集群时，***它们必须将该变更通知到所有磁盘节点***。如果只有一个磁盘节点而且它宕机了，那么集群可以继续路由消息，但是不能做以下的操作：

- 创建队列
- 创建交换器
- 创建绑定
- 添加用户
- 更改权限
- 添加或删除集群节点

也就是说如果集群中的唯一磁盘节点崩溃的话，集群可以保持运行，但是直到该节点恢复前，无法更改任何东西。解决方案是***在集群中设置两个磁盘节点***，保证任何时候至少有一个是可用的，可以在任何时候保存元数据变更。内存节点重启后，会连接到预先配置的磁盘节点下载当前集群元数据拷贝。当添加内存节点时，确保***告知它所有的磁盘节点的地址***（内存节点唯一存储到磁盘的元数据信息是集群中磁盘节点的地址）。

5.4 将节点分布到更多的机器上

RabbitMQ集群对延迟非常敏感，应当只在本地局域网内使用。

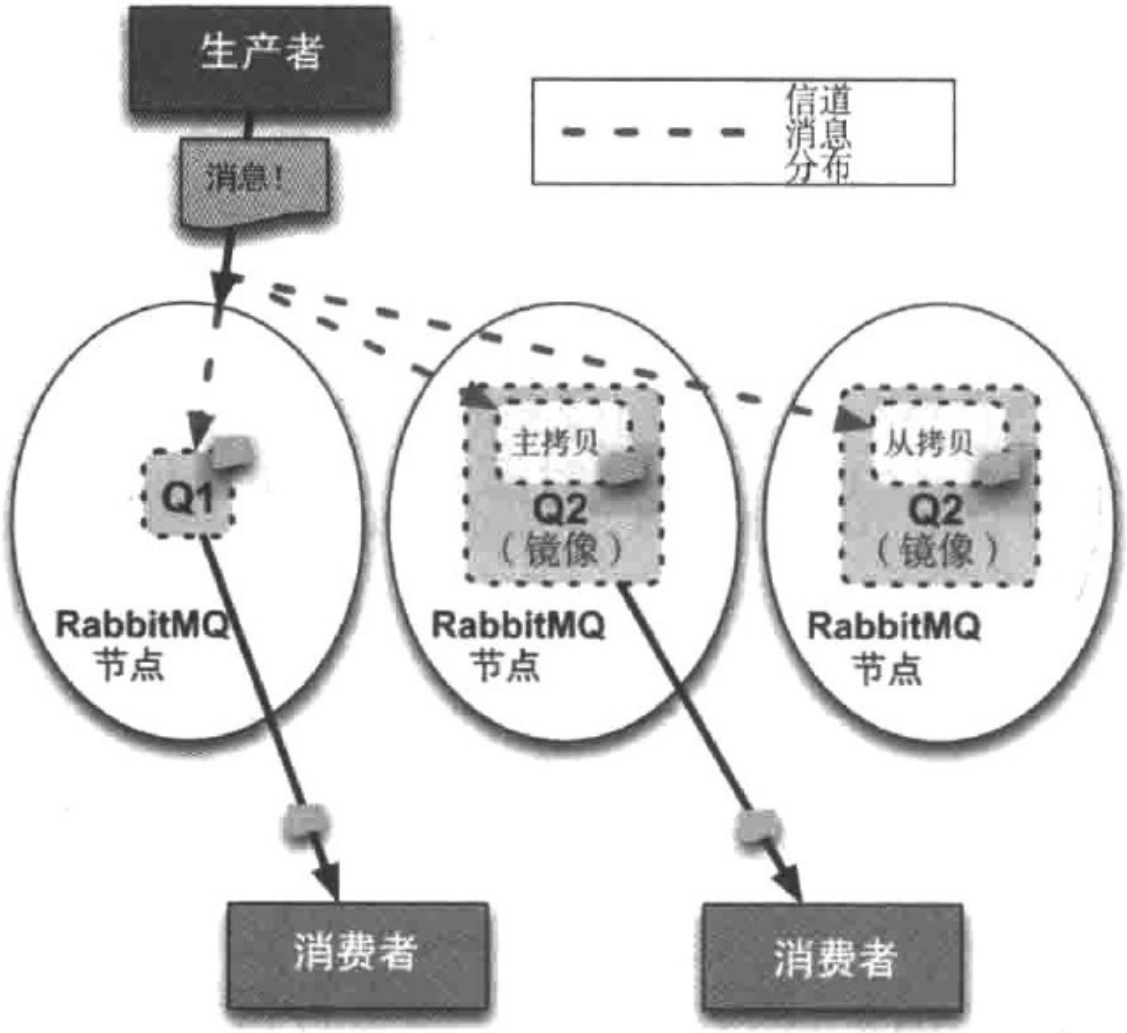
这些机器上的Erlang cookie必须一样，可以把一个服务器上的Erlang cookie复制给其他几个节点。

5.6 镜像队列和保留消息

像普通队列那样，镜像队列的主拷贝仅存在于一个节点上，但与普通队列不同的是，镜像队列在集群中的其他节点上拥有从队列拷贝。一旦队列主节点不可用，***最老的从队列将被选举为新的主队列***。

5.6.2 镜像队列工作原理

在非镜像队列的Rabbit集群中，信道负责将消息路由到合适的队列。当加入镜像队列后，信道仍然做着同样的事情。除了将消息按照路由绑定规则投递到合适的队列之外，它也要将消息投递到镜像队列的从拷贝。



采用发送法确认模式，当处理那些非镜像队列时，在信道根据匹配的绑定规则将消息路由到所有特定的队列后，会收到一个发送方确认消息。当切换到镜像队列时，Rabbit会在队列和队列的从拷贝安全地接收到消息时通知你。但是如果消息在路由到从拷贝前，镜像队列的主拷贝发送故障，并且该从拷贝变成了主拷贝的话，那么发送方确认消息永远不会到达，于是就知道消息可能已经丢失了。

如果镜像队列失去了一个从节点的话，那附加在镜像队列的任何消费者都不会注意到这一点，这是因为它们是附加在主拷贝上的。但是如果托管主拷贝的节点发生故障的话，那么所有该队列的消费者需要重新附加并监听新的队列主拷贝。对于通过故障节点进行连接的消费者（上图中的坐边的消费者）来说没什么困难，因为它们会丢失到节点的TCP连接，在它们重新附加到集群中一个新节点时，会自动选取新的队列主拷贝。但对于那些通过其他节点附加到镜像队列且该节点正常运行的情况（上图中的右边的消费者）RabbitMQ会发送给这些消费者一个消费者取消通知，告知它们已不再附加在队列主拷贝了。

第6章 从故障中恢复

通过负载均衡，不仅可以减少应用程序处理节点故障代码的复杂性，又能确保在集群中连接的平均分布。

6.2 连接丢失和故障转移

当节点发生故障的时候，不能嘉定队列和绑定可以从节点故障中恢复，必须假定在消费的所有队列都附加在该节点之上，并且已不复存在。队列的绑定也一样，但是交换器则不同。如果使用的是内建的Rabbit集群的话可以假设交换器能够幸免于节点故障，因为它们在所有节点都有副本。但是如果使用了后面讲述的主/备模式设置的话，则仍然无法假设交换器可以从故障中恢复。

因此，不论节点故障什么时候发生，在检测到故障并进行重连之后的首要任务是构造交换器、队列和绑定。

队列在集群环境中只存在于某一个节点上，由于在开始消费的时候，应用程序不知道队列在哪个节点上，因此应用程序很可能连接到了集群中的A节点但却从B节点的队列上消费消息，所以当B节点发生故障时，虽然应用程序不会遭受连接错误，但是消费的那个队列却已不复存在。

附录：CentOS安装RabbitMQ镜像集群

只在两台机器上安装两个RabbitMQ，形成集群。

1. 安装erlang环境

1.1 安装依赖文件

```
yum install gcc glibc-devel make ncurses-devel openssl-devel xmlto
```

1.2 下载并解压安装包

```
wget http://erlang.org/download/otp_src_20.3.tar.gz

tar -zxvf otp_src_20.3.tar.gz
```

1.3 配置安装路径，编译代码

```
cd otp_src_20.3.tar.gz

./configure --prefix=/opt/erlang
make
make install
```

1.4 配置Erlang环境变量

```
vi /etc/profile
```

```
# set erlang environment
export PATH=$PATH:/opt/erlang/bin
```

```
source /etc/profile
```

安装完成后可以运行`erl`查看安装结果

2. 安装RabbitMQ

2.1 下载并解压RabbitMQ

```
wget https://dl.bintray.com/rabbitmq/all/rabbitmq-server/3.7.4/rabbitmq-server-generic-unix-3.7.4.tar.xz

xz -d rabbitmq-server-generic-unix-3.7.4.tar.xz
# 解压到/opt目录下
tar -xvf rabbitmq-server-generic-unix-3.7.4.tar.xz -C /opt

# 改名
cd /opt
mv rabbitmq-server-generic-unix-3.7.4 rabbitmq
```

2.2 配置RabbitMQ环境变量

```
vi /etc/profile
```

```
# set erlang environment
export PATH=$PATH:/opt/rabbitmq/sbin
```

```
source /etc/profile
```

2.3 RabbitMQ相关的命令

```
# 启动服务，以后台进程的方式启动
rabbitmq-server -detached

# 查看服务状态
rabbitmqctl status

# 关闭服务，和启动服务相对应，用于停止运行RabbitMQ的Erlang node。
rabbitmqctl stop

# 停止RabbitMQ application，但是Erlang node会继续进行，此命令主要用于优先执行其他管理操作（这些操作需要先停止RabbitMQ application）
rabbitmqctl stop_app

# 重新启动停止的RabbitMQ application
rabbitmqctl start_app
```

2.4 配置网页插件

```
rabbitmq-plugins enable rabbitmq_management
```

2.5 远程访问配置

默认的网页是不允许访问的，需要增加一个用户并且修改一下权限。

```
# 添加用户
rabbitmqctl add_user sherry 123456

# 添加权限，查看3.2.2节
rabbitmqctl set_permissions -p "/" sherry ".*" ".*" ".*"

# 修改用户角色
rabbitmqctl set_user_tags sherry administrator
```

然后就可以远程访问了，浏览器访问`http://hostname:15672`，然后输入用户名和密码就可以了。

3. 配置RabbitMQ集群

3.1 环境准备

两台CentOS7的机器，IP分别为`192.168.1.94`和`192.168.1.91`。

3.2 修改两台机器上的hosts文件

修改hosts文件如下，下面是`192.168.1.94`这台机器的hosts的文件内容：

```
127.0.0.1  node1 localhost localhost.localdomain localhost4 localhost4.localdomain4
::1        node1 localhost localhost.localdomain localhost6 localhost6.localdomain6
192.168.1.94 node1
192.168.1.91 node2
```

`192.168.1.91`也需要如此配置，但是把上面第1行和第2行的node1都换成node2.

注意：一定要在第1行和第2行添加上node1/node2，因为创建集群时，一台RabbitMQ服务器加入另外一个是通过`rabbit@nodex`来访问的，虽然为对应的nodex配置了相应的ip地址，但是RabbitMQ在本机上的默认名称是`rabbit@localhost`，所以如果第一行不加上对应的nodex的话，就会提示连接不上集群。

修改hosts后需要重启机器。

3.3 配置Erlang集群

RabbitMQ的集群是依赖Erlang集群，而Erlang集群是通过cookie进行通信认证的。

所以需要保持集群的所有机器中的.erlang.cookie文件中的cookie值一致，且权限为400。

.erlang.cookie在`~/`目录下，可以通过`cat .erlang.cookie`来查看内容。

3.4 搭建RabbitMQ的一般模式集群

1. 两台机器中选择一个（比如node2），停止它的RabbitMQ服务

```
rabbitmqctl stop_app
```

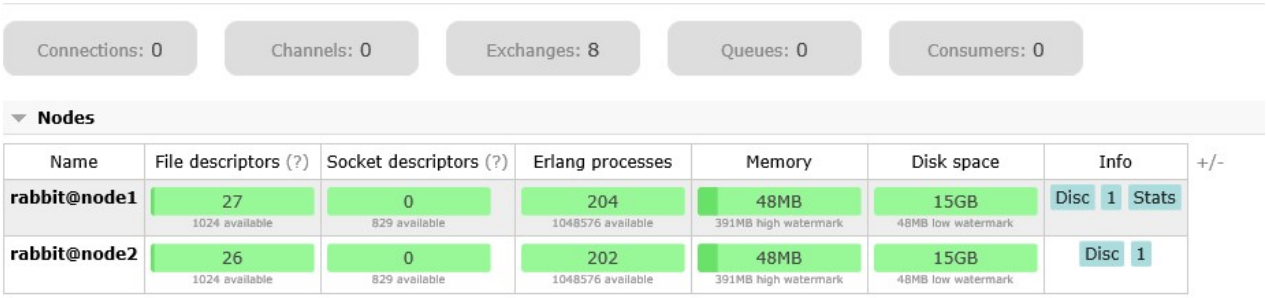
2. 把node2中的RabbitMQ加入到集群中来

```
# 可以添加--ram，这样节点将以RAM节点身份加入集群
rabbitmqctl join_cluster rabbit@node1
```

3. 开启node2中的RabbitMQ服务

```
rabbitmqctl start_app
```

打开网页管理界面查看nodes



3.5 搭建RabbitMQ的镜像高可用模式集群

3.5.1 通过命令行的方式

```
rabbitmqctl set_policy ha-all "^" '{"ha-mode":"all"}'
```

^：为匹配符，只有一个^代表匹配所有。

ha-mode：为匹配类型。

3.5.2 通过web管理界面的方式：

打开web管理界面，点击admin后，再点击右侧的Policies，接着就可以添加Policy了。

User: sherry
Cluster: rabbit@node2 (change)
RabbitMQ 3.6.1, Erlang 20.3

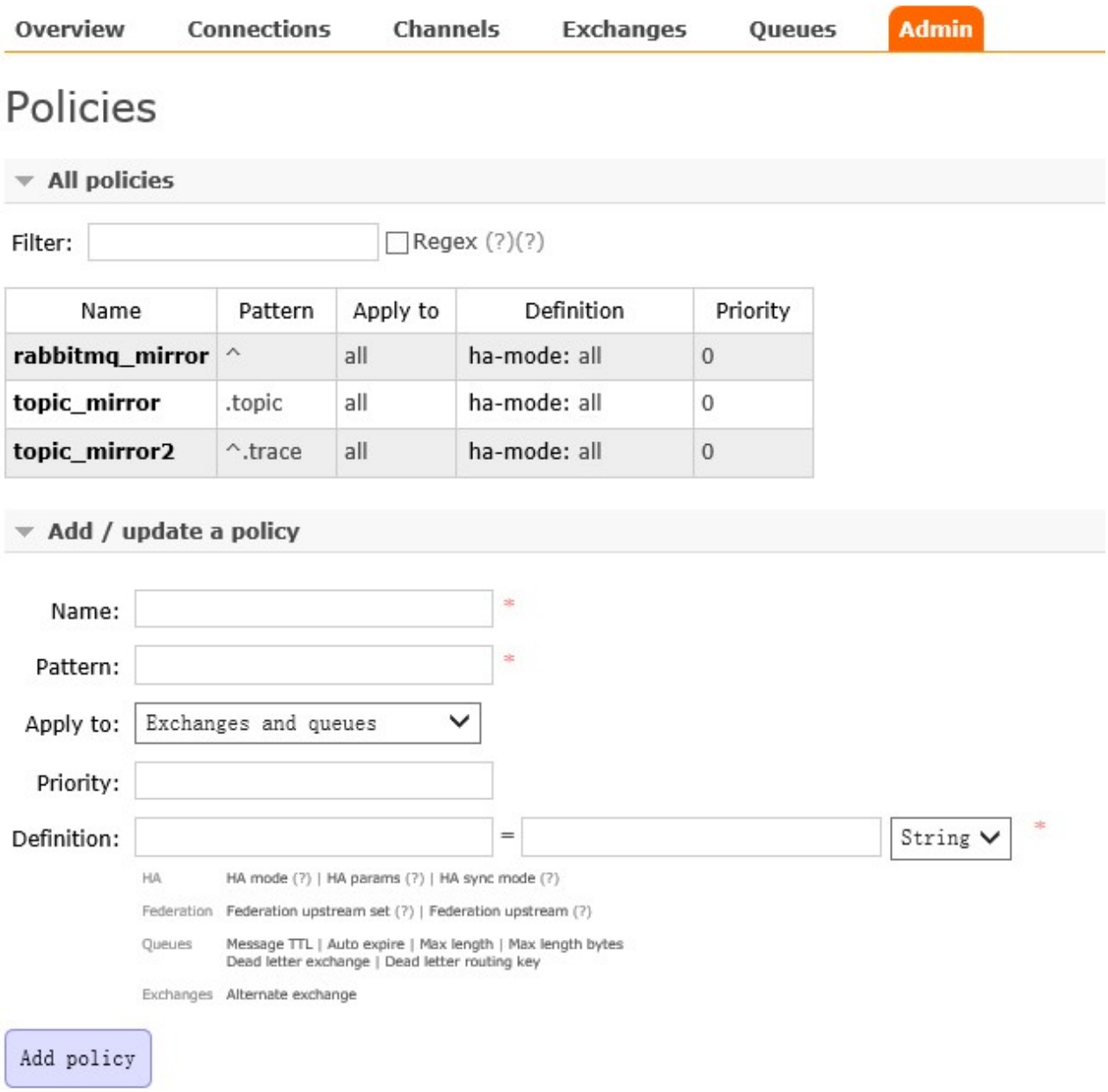
Log out

Users

Virtual Hosts

Policies

3 items, page size up to 100



如图所示：

- Name

  Policy的名字，可以随便取

  Pattern

  匹配的方式：

  - ^：rabbitmq_mirror中的pattern为全部匹配
  - **xx**：topic_mirror中的pattern为包含匹配，即只要Exchanges和Queues的name包含.topic，就会采用这个Policy。
  - **^xx**：topic_mirror2中的pattern为完全匹配，即只有名字为.trace的Exchange和Queue才会采用这个Policy。

- Definition

  第一行必须输入ha-mode，至于第二行输不输则取决于ha-mode的值。

- **all**：所有的节点。
- **exactly**：部分节点，需要配合ha-params参数，此参数为int类型比如3，众多集群中的随机3台机器
- **nodes**：指定，需要配合ha-params参数，此参数为数组类型，比如["rabbit@node1","rabbit@node2"]这样指定为node1与node2这2台机器。

结果如下图所示：