

课题说明

在现在的电商系统中会大量的使用**消息队列**来完成相关的功能。本课程我们是以秒杀场景为切入点来给大家介绍一下在秒杀场景下如何去使用消息队列。主要讲解的是秒杀场景下在使用消

息队列的时候所遇到的问题的以及具体的解决方案。比如：订单未支付的时候如何进行库存的回退，从消息队列角度如何保证秒杀不超卖，秒杀的公平性通过消息队列如何保证，消息的可

靠性如何保证，消息队列的高可用如何实现...

本课程所属体系为：架构师《技术中台篇》，因此偏向于具体技术的讲解，并不会真正去实现一个秒杀系统。具体的秒杀系统的实现在《业务中台篇》有具体的课题讲解。

1 第一章: 秒杀以及常见的消息队列介绍

1.1 秒杀业务介绍

1.1.1 秒杀简介

秒杀介绍

所谓“秒杀”，就是网络卖家发布一些超低价格的商品，所有买家在同一时间网上抢购的一种销售方式。通俗一点讲就是网络商家为促销等目的组织的网上限时抢购活动。由于商品价格低廉，往往一上架就被抢购一空，有时只用一秒钟。秒杀活动结束的两种限制：库存限制、时间限制。

特点介绍

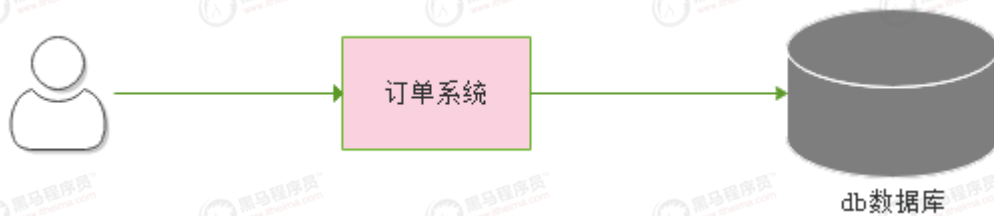
秒杀具有以下几个业务特点:

- 定时触发，流量在瞬间突增
- 秒杀请求中常常只有部分能够成功

- 秒杀商品数量往往有限，不能超卖，但能接受少卖
- 不要求立即返回真实下单结果

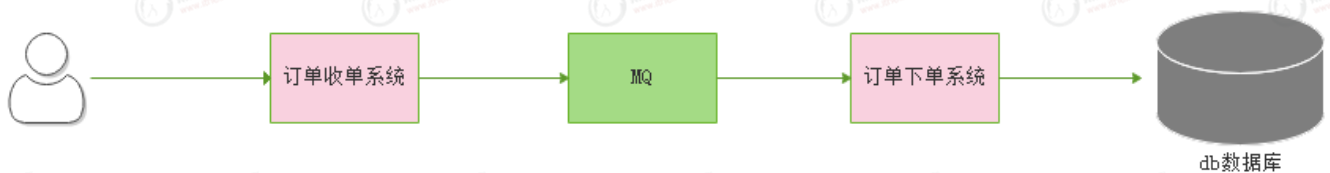
由于秒杀业务的特殊性--短时超高并发，因此我们不能按照传统的交易场景进行设计。传统交易场景下，对于用户的下单请求一般都是同步处理，即同步落库持久化，并同步返回收单结果。

如果我们对秒杀订单采用同步持久化的做法，则系统的吞吐量将基本依赖 DB 的性能，这在成本上、性能上都有较大压力。



因此，我们要在尽量提高系统收单入口吞吐量的同时降低系统开发部署的成本。**"不要求立即返回真实下单结果"**，也就是不需要立即持久化，换言之也就是业务流程**"异步化"**。明确了流程

可以异步化，解决的手段就多了。利用缓存、队列、线程池都能实现业务的异步化。如下所示：



1.1.2 秒杀流程介绍

通过对秒杀核心业务流程进行异步化，我们能够将主流程分为收单、下单两个阶段，业务流程概括起来如下：

秒杀流程-收单

- 用户访问秒杀入口，将秒杀请求提交给秒杀平台收单网关，平台对秒杀请求进行前置校验
- 校验通过后，将下单请求通过缓存/队列/线程池等中间层进行提交，在投递完成同时就给用户返回“排队中”
- 对于前置校验失败的下单请求同步返回秒杀下单失败

秒杀流程-下单

下单流程中，平台的压力通过中间层的缓冲其实已经小了很多，之所以会很小，原因有两方面：

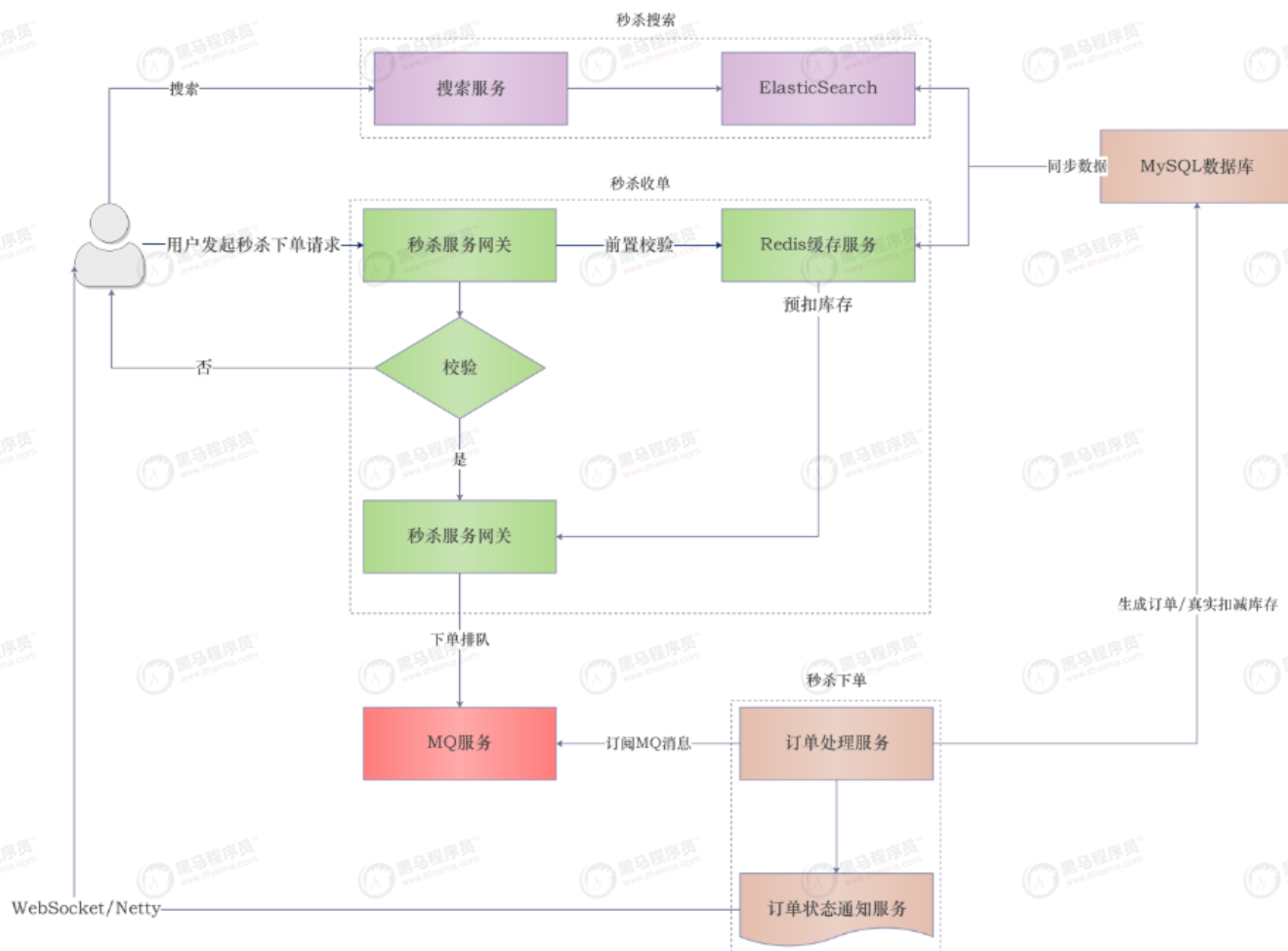
- 1、一方面是因为在用户下单的同步校验过程中就过滤掉了部分非法请求
- 2、请求在中间层进行了缓存，等待下游的业务系统（秒杀下单）慢慢消化，尽可能减少流量对平台持久层的冲击，这里其实就体现了中间层“削峰填谷”的特点。

基于上述前提，我们简单总结下秒杀下单部分的业务逻辑。

- 秒杀订单服务获取中间层的下单请求，进行真实的下单前校验，这里主要进行库存的真实校验
- 扣减库存（或称锁库存）成功后，发起真实的下单操作。扣减库存（锁库存）与下单操作一般在一个事务域中下单成功后，平台往往会发起消息推送，告知用户下单成功，并引导用户进行支付操作
- 用户一段时间（如：30mins）没有支付，则订单作废，库存恢复，给其他排队中的用户提供购买机会
- 如果用户支付成功，则订单状态更新，订单流转到其他子系统，如：物流系统对该支付成功的处理中订单进行发货等后续处理

秒杀架构图

基于以上的几点我们本次所设计的秒杀架构图如下所示：



- 1、用户录入秒杀数据以后，我们需要将秒杀数据同步到Redis和ElasticSearch中
- 2、用户搜索的秒杀商品是从ElasticSearch中进行搜索
- 3、进行秒杀收单操作
- 4、进行秒杀下单操作

1.2 常见消息比对

通过对秒杀架构的了解，我们发现在秒杀架构中mq起到了极大的作用。本课题主要研究的就是在秒杀业务中，mq这一小段如何保证秒杀的"公平和不超卖"。MQ相关的产品有：

ActiveMQ, RabbitMQ, RocketMQ, Kafka, Zeromq。目前市面上常用的消息队列产品：RabbitMQ, Kafka, RocketMQ。因此我们首先来比对一下这几个MQ之前的差别：

比较方向	RabbitMQ	Kafka	RocketMQ
资料文档	多 (有一些不错的书, 网上资料多)	中 (有kafka作者自己写的书, 网上资料也有一些)	少 (没有专门写rocketmq的书, 网上的资料良莠不齐, 官方文档很简洁, 但是对技术细节没有过多的描述。)
开发语言	Erlang	Scala	java
支持的协议	AMQP	自定义的一套协议	自定义的一套协议
消息存储	内存、磁盘 (支持少量的消息堆积)	磁盘 (支持大量的消息堆积)	磁盘 (支持大量的消息堆积)
事务消息	支持	支持	支持
管理界面	好	一般	无
消息重复	支持at least once、at most once	支持at least once、at most once	支持at least once
吞吐量	单机万级	单机十万级	单机万级
顺序消息	在满足一些前提的情况下, 支持消息的顺序性	支持	支持
消息确认	支持	支持	支持

总结:

1、一般的业务系统要引入MQ, 最早大家都用ActiveMQ, 但是现在确实大家用的不多了, 没经过大规模吞吐量场景的验证, 社区也不是很活跃。

2、后来大家开始用RabbitMQ, 但是erlang语言阻止了大量的java工程师去深入研究和掌控他, 对公司而言, 几乎处于不可控的状态。但由于是开源的, 并且也比较稳定, 活跃度也高, 所以使

用很广泛

3、现在很多的公司也开始使用RocketMQ, 由于社区活跃度不高, 资料比较少。所以学习成本比较大, 因此要使用RocketMQ要考虑社区万一突然黄掉的风险。所以中小型公司, 技术实力较

为一般, 技术挑战不是特别高, 用RabbitMQ是不错的选择; 大型公司, 基础架构研发实力较强, 用RocketMQ是很好的选择。

4、如果是大数据领域的实时计算、日志采集等场景, 用Kafka是业内标准的, 绝对没问题, 社区活跃度很高, 绝对不会黄, 何况几乎是全世界这个领域的事实性规范

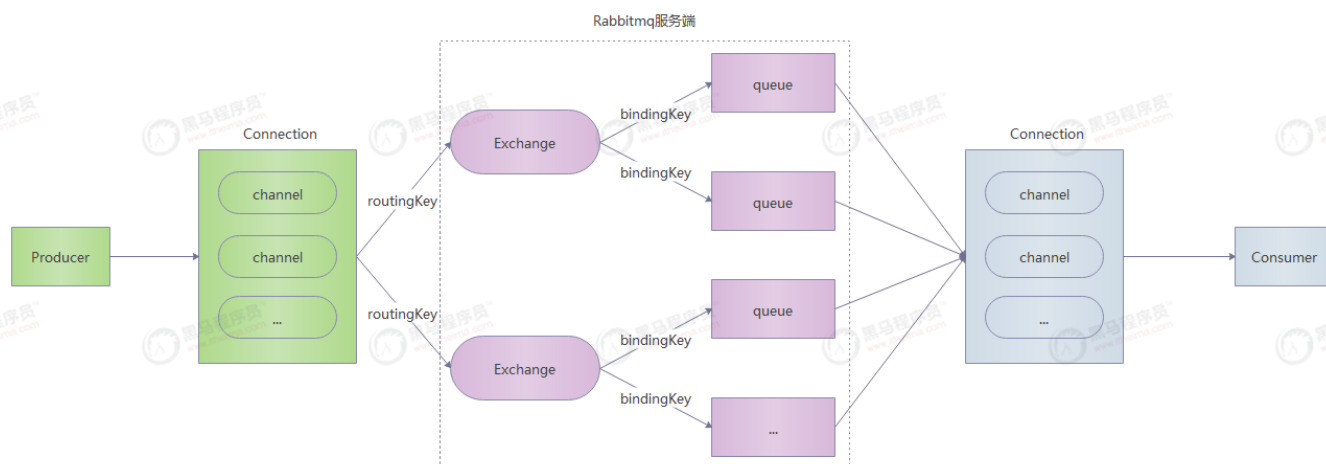
2 第二章: RabbitMQ在秒杀场景下应用

2.1 RabbitMQ基础知识回顾

2.1.1 RabbitMQ架构回顾

RabbitMQ是实现了高级消息队列协议（AMQP）的开源消息代理软件（亦称面向消息的中间件）。RabbitMQ服务器是用[Erlang](#)语言编写的。它的实现架构

如下图所示：



各个组件介绍：

Exchange：消息交换机,它指定消息按什么规则,路由到哪个队列

Queue:消息的载体,每个消息都会被投到一个或多个队列

Binding:绑定，它的作用就是把exchange和queue按照路由规则绑定起来

Routing Key:路由关键字,exchange根据这个关键字进行消息投递

Producer:消息生产者,就是投递消息的程序

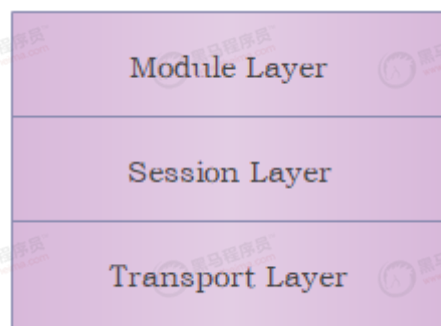
Consumer:消息消费者,就是接受消息的程序

Channel:消息通道,在客户端的每个连接里,可建立多个channel

2.1.2 AMQP协议介绍

AMQP协议基本介绍

目前Rabbitmq最新的版本默认支持的是AMQP 0-9-1，该协议总共包含了3部分：



Module Layer： 位于协议的最高层，**主要定义了一些供客户端调用的命令**，客户端可以利用这些命令实现自定义的业务逻辑。

例如，客户端可以是使用Queue.Declare命令声明一个队列或者使用Basic.Consume订阅消费一个队列中的消息。

Session Layer： 位于中间层，主要负责将客户端的命令发送给服务端，在将服务端的应答返回给客户端，主要为客户端与服务器之间的通信提供可靠性的同步机制和错误处理。

Transport Layer： 位于最底层，主要传输二进制数据流，提供帧的处理、信道的复用、错误检查和数据表示等。

AMQP生产者流转过程

接下来我们就来了解一些AMQP协议中生产者发送消息时所涉及到的相关命令，以及在发送数据的时候命令的传输的整体过程。

我们以示例代码itheima-edu-springboot-rabbitmq-producer(producerMessageTransport)方法为例来说明一下，命令的传输流程(通过抓包工具进行抓取，

选择虚拟机网卡)如下图所示：

15	0.319681	192.168.23.1	192.168.23.131	AMQP	62 Protocol-Header 0-9-1
17	0.322007	192.168.23.131	192.168.23.1	AMQP	570 Connection.Start
18	0.326963	192.168.23.1	192.168.23.131	AMQP	499 Connection.Start-Ok
19	0.327734	192.168.23.131	192.168.23.1	AMQP	74 Connection.Tune
20	0.331073	192.168.23.1	192.168.23.131	AMQP	74 Connection.Tune-Ok
21	0.331886	192.168.23.1	192.168.23.131	AMQP	70 Connection.Open vhost=
23	0.332472	192.168.23.131	192.168.23.1	AMQP	67 Connection.Open-Ok
24	0.341289	192.168.23.1	192.168.23.131	AMQP	67 Channel.Open
25	0.342579	192.168.23.131	192.168.23.1	AMQP	70 Channel.Open-Ok
26	0.354822	192.168.23.1	192.168.23.131	AMQP	96 Exchange.Declare x=fanout.exchange
27	0.355765	192.168.23.131	192.168.23.1	AMQP	66 Exchange.Declare-Ok
28	0.357557	192.168.23.1	192.168.23.131	AMQP	86 Queue.Declare q=fanout.queue
29	0.358468	192.168.23.131	192.168.23.1	AMQP	87 Queue.Declare-Ok q=fanout.queue
30	0.368262	192.168.23.1	192.168.23.131	AMQP	183 Queue.Bind q=fanout.queue x=fanout.exchange bk=
31	0.362144	192.168.23.131	192.168.23.1	AMQP	66 Queue.Bind-Ok
32	0.365442	192.168.23.1	192.168.23.131	AMQP	180 Basic.Publish x=fanout.exchange rk= Content-Header type=text/plain Content-Body (text/plain)

当客户端与Broker建立连接的时候，客户端会向Broker发送一个Protocol Header 0-9-1的报文头，以此通知Broker本次交互才采用的是AMQP 0-9-1协议，紧接着Broker返回Connection.Start来

建立连接，在连接的过程中涉及Connection.Start/.Start-OK、Connection.Tune/.Tune-OK、Connection.Open/.Open-OK这6个命令的交互。连接建立以后需要创建通道，会使用到

Channel.Open，Channel.Open-OK命令，在进行交换机声明的时候需要使用到Exchange.Declare以及Exchange.Declare-OK的命令。以此类推，在声明队列以及完成队列和交换机的绑定的时候

都会使用到指定的命令来完成。在进行消息发送的时候会使用到Basic.Publish命令完成，这个命令还包含了Content-Header和Content-Body。Content Header里面包含的是消息体的属性，

Content-Body包含了消息体本身。

由于我们本次演示的代码是spring boot和rabbitmq进行整合以后的代码，没有涉及到通道的关闭以及连接的释放，因此看不到关闭通道以及连接的相关命令传输过程。

AMQP消费者流转过程

接下来我们就来了解一些AMQP协议中消费消费时所涉及到的相关命令，以及在消费消息的时候命令的传输的整体过程。

我们以示例代码itheima-edu-springboot-rabbitmq-consumer(consumerMessageTransport)方法为例来说明一下，命令的传输流程如下图所示：

6	0.009757	192.168.23.131	192.168.23.1	AMQP	570 Connection.Start
7	0.014938	192.168.23.1	192.168.23.131	AMQP	499 Connection.Start-Ok
8	0.015857	192.168.23.131	192.168.23.1	AMQP	74 Connection.Tune
9	0.018868	192.168.23.1	192.168.23.131	AMQP	74 Connection.Tune-Ok
10	0.019855	192.168.23.1	192.168.23.131	AMQP	70 Connection.Open vhost=
12	0.020498	192.168.23.131	192.168.23.1	AMQP	67 Connection.Open-Ok
13	0.034133	192.168.23.1	192.168.23.131	AMQP	67 Channel.Open
14	0.035800	192.168.23.131	192.168.23.1	AMQP	70 Channel.Open-Ok
15	0.048349	192.168.23.1	192.168.23.131	AMQP	96 Exchange.Declare x=fanout.exchange
16	0.049110	192.168.23.131	192.168.23.1	AMQP	66 Exchange.Declare-Ok
17	0.050779	192.168.23.1	192.168.23.131	AMQP	86 Queue.Declare q=fanout.queue
18	0.051733	192.168.23.131	192.168.23.1	AMQP	87 Queue.Declare-Ok q=fanout.queue
19	0.053237	192.168.23.1	192.168.23.131	AMQP	183 Queue.Bind q=fanout.queue x=fanout.exchange bk=
20	0.054498	192.168.23.131	192.168.23.1	AMQP	66 Queue.Bind-Ok
21	0.062343	192.168.23.1	192.168.23.131	AMQP	86 Queue.Declare q=fanout.queue
22	0.063252	192.168.23.131	192.168.23.1	AMQP	87 Queue.Declare-Ok q=fanout.queue
23	0.067710	192.168.23.1	192.168.23.131	AMQP	86 Queue.Declare q=fanout.queue
24	0.068428	192.168.23.131	192.168.23.1	AMQP	87 Queue.Declare-Ok q=fanout.queue
25	0.069332	192.168.23.1	192.168.23.131	AMQP	73 Basic.Qos
26	0.069941	192.168.23.131	192.168.23.1	AMQP	66 Basic.Qos-Ok
27	0.072540	192.168.23.1	192.168.23.131	AMQP	87 Basic.Consume q=fanout.queue
28	0.073955	192.168.23.131	192.168.23.1	AMQP	262 Basic.Consume-Ok Basic.Deliver x=fanout.exchange rk= Content-Header type=text/plain Content-Body (text/plain)
29	0.089033	192.168.23.1	192.168.23.131	AMQP	75 Basic.Ack

我们发现消费者消费消息的时候，所涉及到的命令和生成者大部分都是相同的。在原有的基础之上，多个几个命令：Basic.Qos/.Qos-OK以及Basic.Consume和Basic.Consume-OK。其中

Basic.Qos/.Qos-OK这两个命令主要用来确认消费者最大能保持的未确认的消息数时使用。Basic.Consume和Basic.Consume-OK这两个命令主要用来进行消息消费确认。

AMQP命令概览

见附录资料中的"01-AMQP协议中定义的常用命令.png"图片。

2.1.3 交换机的类型

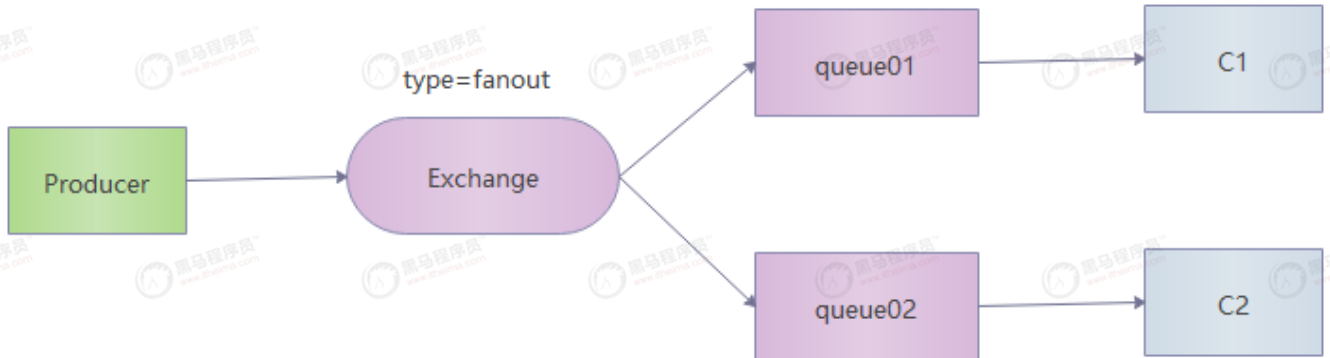
fanout

扇型交换机 (funout exchange) 将消息路由给绑定到它身上的所有队列，而**不理睬绑定的路由键**。如果 N 个队列绑定到某个扇型交换机上，当有消息发送给此扇型交换机时，交换机会将消息的拷贝分别发送给这所有的 N 个队列。扇型用来交换机处理消息的广播路由 (broadcast routing)。

因为扇型交换机投递消息的拷贝到所有绑定到它的队列，所以他的应用案例都极其相似：

- 大规模多用户在线游戏可以使用它来处理排行榜更新等全局事件
- 体育新闻网站可以用它来近乎实时地将比分更新分发给移动客户端
- 分发系统使用它来广播各种状态和配置更新
- 在群聊的时候，它被用来分发消息给参与群聊的用户

扇型交换机图例：

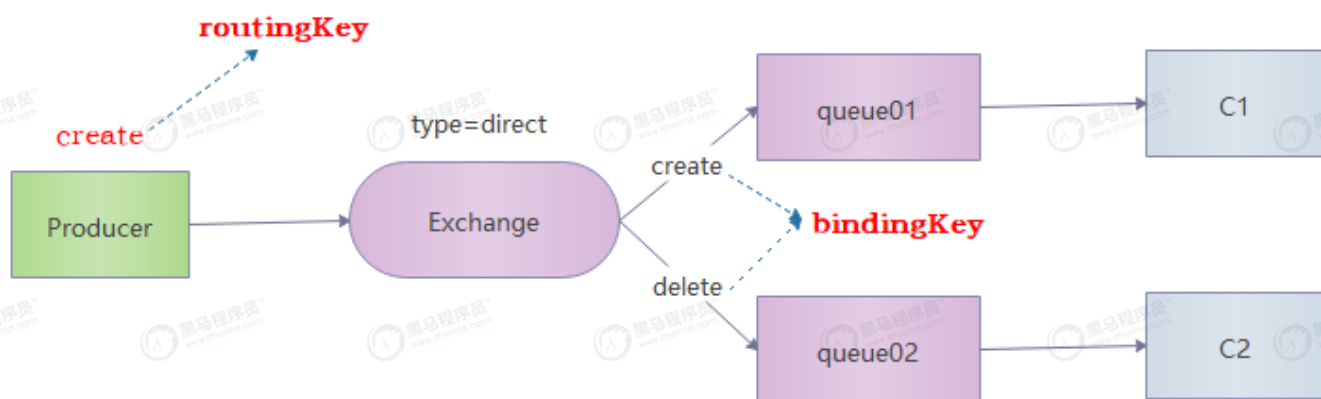


测试代码：

- RabbitmqFanoutExchangeConfiguration配置类
- RabbitmqFanoutExchangeConsumer消费者类
- fanoutExchangeMessageTransport生产者方法

direct

直连型交换机（direct exchange）是根据消息携带的路由键（routing key）将消息投递给对应绑定键的队列。直连交换机主要用来处理消息的单播路由（unicast routing）（尽管它也可以处理多播路由）。下边介绍它是如何工作的：



测试代码：

- RabbitmqDirectExchangeConfiguration配置类
- RabbitmqDirectExchangeConsumer消费者类
- directExchangeMessageTransport生产者方法

黑马程序员
www.it-ebooks.com

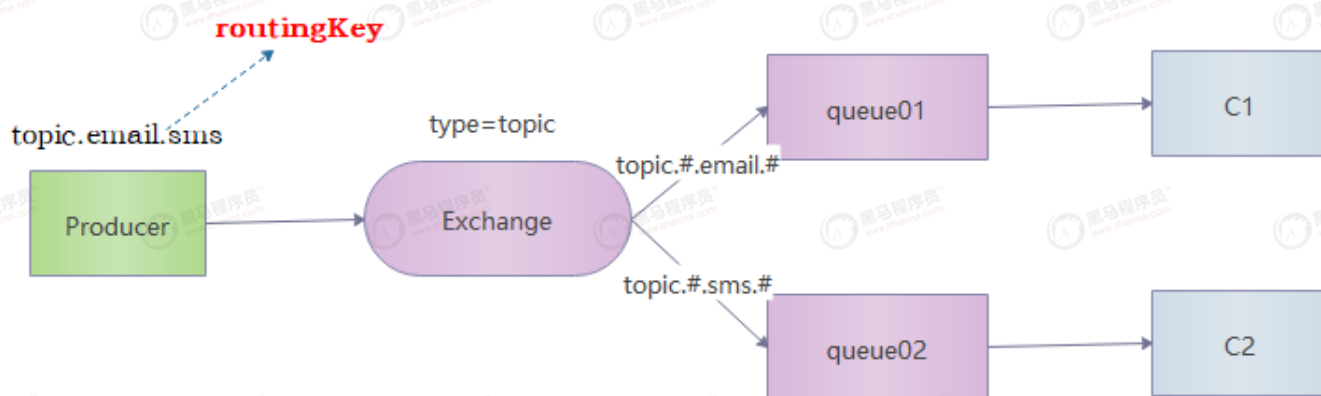
topic

前面提到的 direct 规则是严格意义上的匹配，换言之 Routing Key 必须与 Binding Key 相匹配的时候才将消息传送给 Queue.而Topic 的路由规则是一种模糊匹配，可以通过通配符满足一部分规则就可以传送。

它的约定是：

- 1) 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）
- 2) 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"topic.email"、"topic.sms"、

binding key 与 routing key 一样也是句点号 "." 分隔的字符串。下面介绍它是如何工作的：



测试代码：

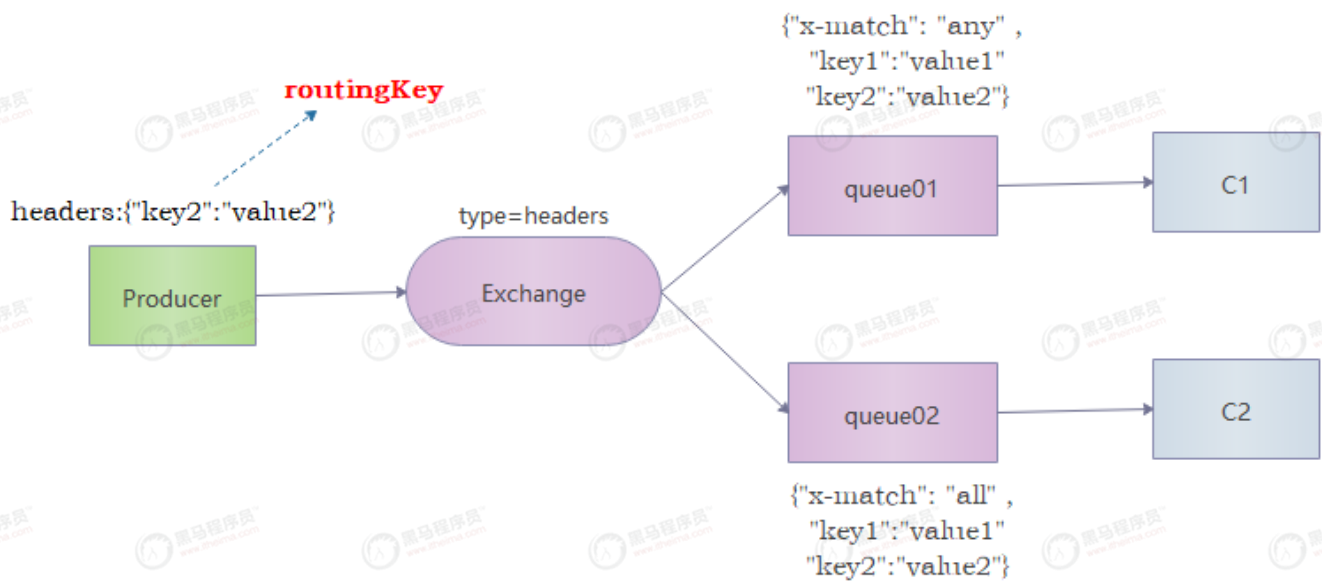
- RabbitmqTopicExchangeConfiguration配置类
- RabbitmqTopicExchangeConsumer消费者类
- topicExchangeMessageTransport生产者方法

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**。在绑定队列和交换机的时候指定一组键

值对，当发生消息到交换机时，Rabbitmq 会获取到该消息的 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队列和交换机绑定时指定的键值对，如果完全匹配则消

息会路由到该队列，否则不会路由到该队列。headers 类型的交换机性能会很差，所以在生产环境中基本不会使用。下面介绍它是如何工作的：



“x-match”的匹配规则（就是上面介绍的all、any的规则）

测试代码：

- RabbitmqHeadersExchangeConfiguration配置类
- RabbitmqHeadersExchangeConsumer消费者类
- headersExchangeMessageTransport生产者方法

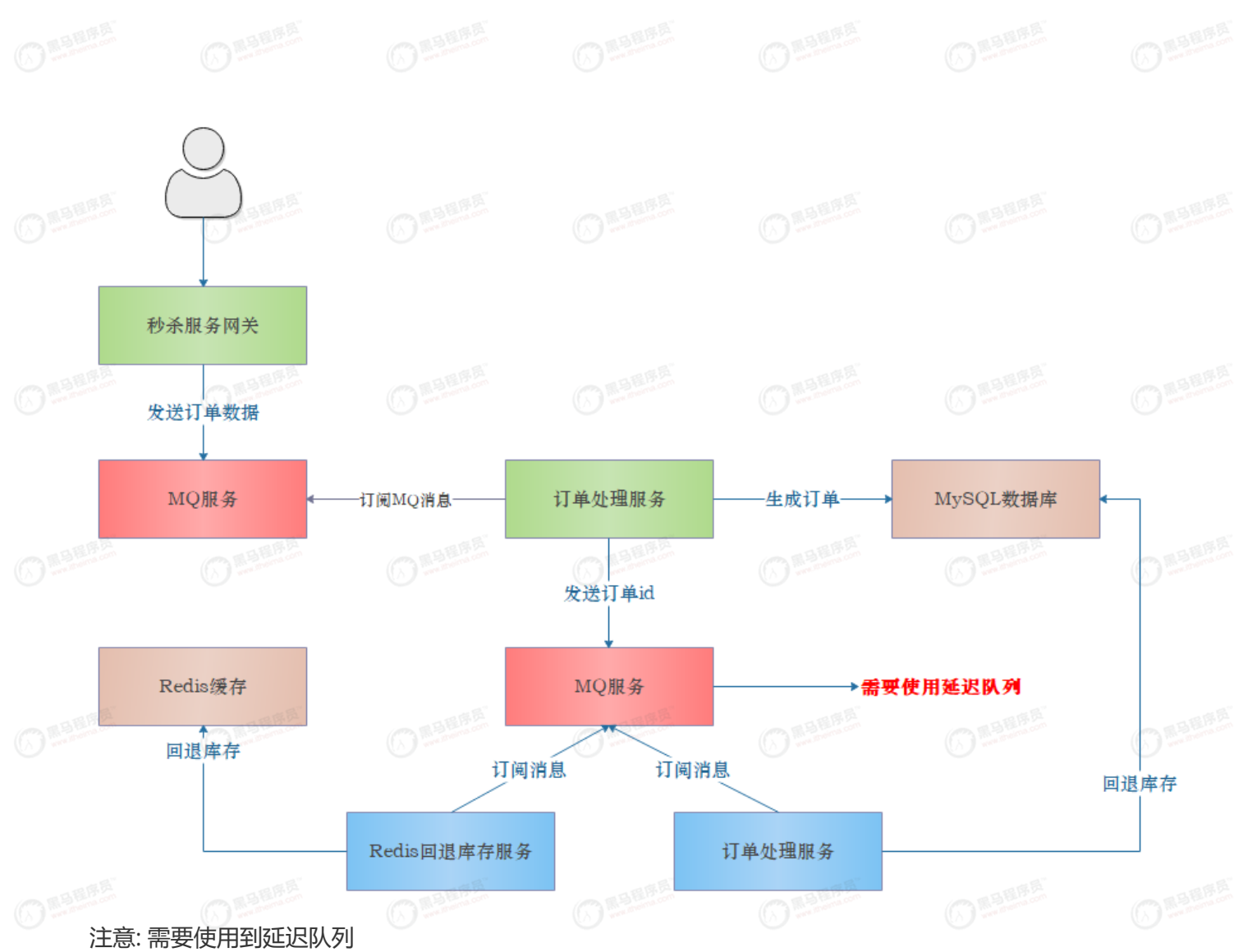
小结：Rabbitmq中的这四种交换机，使用的较多的是direct。主要是基于两个原因：1. direct 交换机可以实现fanout的功能 2. 性能排序：fanout > direct >> topic。

2.2 RabbitMQ在订单未支付时库存回退

当用户秒杀成功以后，就需要引导用户去订单页面进行支付。如果用户在规定的时间内(30分钟)，没有完成订单的支付，此时我们就需要进行库存的回退操作。

2.2.1 库存回退架构

回退库存的架构如下图所示：



2.2.2 过期时间

目前有两种方法可以设置消息的TTL:

- 1、通过队列的属性设置，队列中所有的消息都有相同的过期时间
- 2、是对消息本身进行单独的设置，每条消息的TTL可以不同。

如果两种方法一起使用，则消息的TTL以两者之间较小的那个数值为准。消息在队列中的生存时间一旦超过设置的TTL值时，就会变成“死信”。

设置消息的TTL

在发送消息的时候可以直接通过消息属性来设置消息的过期时间，代码如下所示：

```
1 // 测试发送消息时设置发送消息的过期时间
2 private static void sendPreMessageTTL() {
3     // void convertAndSend(String exchange, String routingKey, Object
4     message, MessagePostProcessor messagePostProcessor)
5     rabbitTemplate.convertAndSend("direct.exchange" , "create" , "测试延
6     迟消息发送，为每一个消息去设置过期时间" , (message) -> {
7         MessageProperties messageProperties =
8         message.getMessageProperties(); // 获取消息属性对象
9         messageProperties.setExpiration("30000"); // 设置消息的过期时间为30秒
10        return message ;
11    });
12 }
```

如果消息没有及时消费，那么经过30秒以后，消息变成死信，Rabbitmq会将这个消息直接丢弃。

设置队列的TTL

我们也可以直接通过队列的属性设置消息的过期时间，队列中所有的消息都有相同的过期时间，代码如下所示：

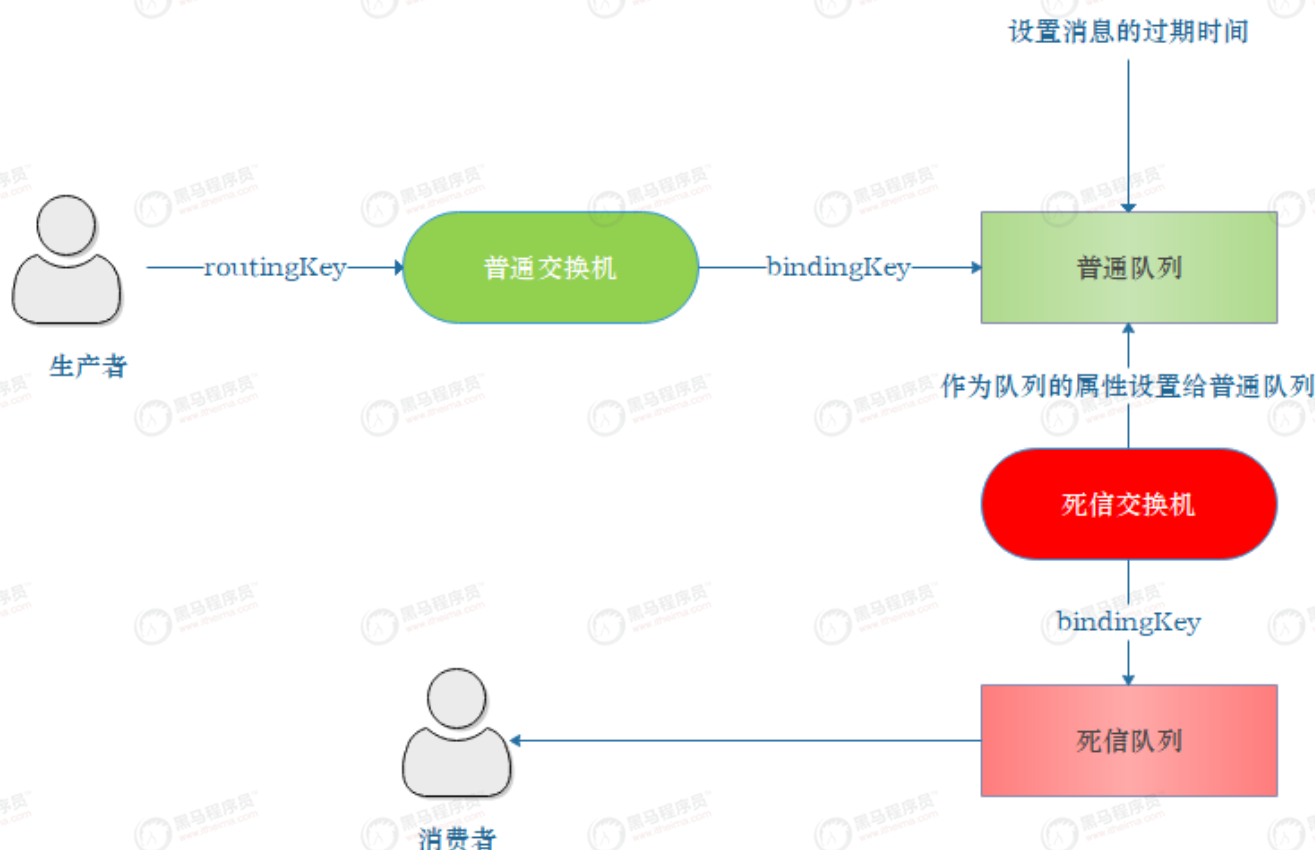
```
1 // 声明队列
2 @Bean(name = "direct.queue_02")
3 public Queue commonQueue02() {
4     QueueBuilder queueBuilder = QueueBuilder.durable("direct.queue_02");
5     queueBuilder.ttl(30000) ; // 设置队列消息的过期时间，为30
    秒
6     return queueBuilder.build() ;
7 }
```

2.2.3 死信队列

当一个消息变成死信了以后，默认情况下这个消息会被mq删除。如果我们给队列指定了"死信交换机"（DLX: Dead-Letter-Exchange），那么此时这个消息就会转发到死信交换机，进而被与死信交换机绑定的队列（死信队列）进行消费。从而实现了延迟消息发送的效果。

原理介绍

具体的原理如下图所示：



一个消息成为死信常见情况：

- 1、消息过期
- 2、队列达到最大长度（可以通过x-max-length参数来指定队列的长度，如果不指定，可以认为是无限长）

具体实现

接下来我们就来演示一下死信队列的使用，使用死信队列来实现消息的延迟发送，具体的代码实现如下所示：

- 1、声明死信交换机、死信队列、死信交换机和死信队列的绑定

```

1 // 声明死信交换机
2 @Bean(name = "dlx.exchange")
3 public Exchange dlxExchange() {
4     return
5     ExchangeBuilder.directExchange("dlx.exchange").durable(true).build() ;
6 }
7 // 声明死信队列
8 @Bean(name = "dlx.queue")
9 public Queue dlxQueue() {
10     return QueueBuilder.durable("dlx.queue").build() ;
11 }
12
13 // 完成死信队列和死信交换机的绑定
14 @Bean
15 public Binding dlxQueueBindToDlxExchange(@Qualifier(value =
16     "dlx.exchange") Exchange exchange , @Qualifier(value = "dlx.queue")
17     Queue queue) {
18     return
19     BindingBuilder.bind(queue).to(exchange).with("delete").noargs() ;
20 }

```

2、将死信队列作为普通队列的属性设置过去

```

1 // 声明队列
2 @Bean(name = "direct.queue_02")
3 public Queue commonQueue02() {
4     QueueBuilder queueBuilder =
5     QueueBuilder.durable("direct.queue_02");
6     queueBuilder.deadLetterExchange("dlx.exchange") ; // 将死信交换机作
7     // 为普通队列的属性设置过去
8     queueBuilder.deadLetterRoutingKey("delete") ; // 设置消息的
9     routingKey
10    // queueBuilder.ttl(30000) ; // 设置队列消息的
11    // 过期时间，为30秒
12    // queueBuilder.maxLength(2) ; // 设置队列的最大
13    // 长度
14    return queueBuilder.build() ;
15 }

```

3、消费端进行同样的设置，并且指定消费死信队列

```
1 @Component
2 public class RabbitmqDlxQueueConsumer {
3
4     // 创建日志记录器
5     private static final Logger LOGGER =
6     LoggerFactory.getLogger(RabbitmqDlxQueueConsumer.class) ;
7
8     @RabbitListener(queues = "dlx.queue")
9     public void dlxQueueConsumer(String msg) {
10         LOGGER.info("dlx queue msg is : {} " , msg );
11     }
12 }
```

4、启动程序进行测试

2.3 RabbitMQ消息可靠性传输

在我们的业务系统中，一旦使用到了消息队列，我们就必须考虑消息的丢失问题。比如在秒杀业务中，一旦消息丢失了对我们用户而言就是不公平的。

2.3.1 第一种情况

场景描述

生产者已经将消息发送给了队列，但是此时消费者还没能及时对消息进行消费，这个时候指定的队列主机宕机了，这样存储在队列的消息也会丢失。

解决方案

解决方案：对消息进行**持久化**操作。当对消息进行持久化操作以后，这个消息一旦被发送到mq中的某一个队列，那么此时Rabbitmq会立马将消息进行持久化。

注意：spring boot和rabbitmq进行整合以后，默认消息的存储就是持久化方式。我们可以将所有的消息都设置为持久化，但是这样会影响Rabbitmq的性能。因为我们需要将消息写入到

内存的同时还需要将消息写入到磁盘。对于可靠性不是那么高的消息可以不采用持久化处理，以提高整体系统的吞吐量。

消息默认时持久化模式：

```
Message 1
The server reported 0 messages remaining.
Exchange    direct.exchange
Routing Key  create
Redelivered  0
Properties   priority: 0
             delivery_mode: 2
             headers:
               content_encoding: UTF-8
               content_type: text/plain
Payload      28 bytes
Encoding: string
direct.exchange 测试数据
```

设置消息为非持久化模式：

```
1 // 测试Direct类型的交换机
2 private static void directExchangeMessageTransport() {
3     rabbitTemplate.convertAndSend("direct.exchange" , "create" , "direct
exchange 测试数据" , (message) -> {
4         // spring boot和rabbitmq整合以后，默认消息是会被持久化的，我们可以将消息
的持久化方式设置为不进行持久化
5         message.getMessageProperties().setDeliveryMode(MessageDeliveryMode.NON_
PERSISTENT);
6         return message ;
7     });
8 }
```

Message 1

The server reported 0 messages remaining.

Exchange	direct.exchange
Routing Key	create
Redelivered	0
Properties	priority: 0 delivery_mode: 1 headers: content_encoding: UTF-8 content_type: text/plain
Payload	direct exchange 测试数据
Encoding: string	

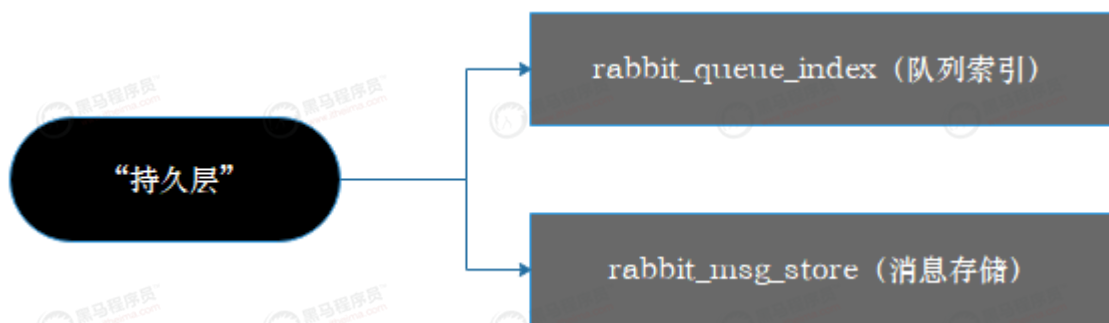
相关知识：消息存储机制

不管是持久化的消息还是非持久化的消息都可以被写入到磁盘。

1、持久化的消息在到达队列时就被写入到磁盘，并且如果可以，持久化的消息也会在内存中保存一份备份，这样可以提高一定的性能，当内存吃紧的时候会从内存中清除。

2、非持久化的消息一般只保存在内存中，在内存吃紧的时候会被写入到磁盘中，以节省内存空间。

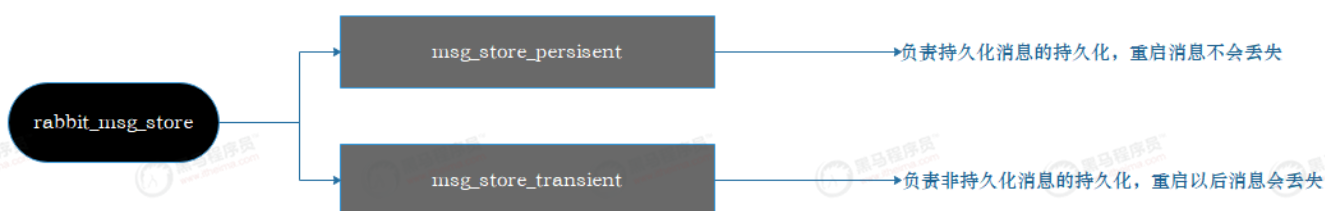
这两种类型的消息的落盘处理都在RabbitmqMQ的"持久层"中完成。持久层的组成如下所示：



rabbit_queue_index: 负责维护队列中的落盘消息的信息，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。每一个队列都有与之对应的一个rabbitmq_queue_index。

rabbit_msg_store: 负责消息的存储，它被所有的队列共享，在**每个节点中有且只有一个**。

rabbit_msg_store可以在进行细分：



在容器中默认这些信息是通过/var/lib/rabbitmq/mnesia/rabbit@977b5f791952这个路径下的3个文件夹进行存储：

```
drwxr-xr-x. 5 root root 110 Jun 13 08:10 .
drwxr-xr-x. 3 root root 39 Jun 11 09:47 ..
-rw-r--r--. 1 root root 1 Jun 13 08:10 .vhost
drwxr-xr-x. 2 root root 19 Jun 13 08:10 msg_store_persistent
drwxr-xr-x. 2 root root 19 Jun 13 08:10 msg_store_transient
drwxr-xr-x. 8 root root 204 Jun 13 08:08 queues
-rw-r--r--. 1 root root 5464 Jun 13 08:10 recovery.data
```

消息可以存储在rabbit_queue_index中也可以存储在rabbit_msg_store中。最佳的配置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。这个消息的界定

可以通过queue_index_embed_msgs_below来配置，默认大小为4096，单位为B。注意这里的消息大小是指消息体、属性以及headers整体的大小。当一个消息小于设定的大小阈值时就可以

存储在rabbit_queue_index中，这样可以得到性能上的优化。这种存储机制是在Rabbitmq3.5 版本以后引入的，该优化提高了系统性能10%左右。

测试：删除queues文件夹，重启rabbitmq服务，发现此时消息已经丢失了。

通过如下的命令我们就可以查看到Rabbitmq在运行的时候的一些运行参数：

```
1 | rabbitmqctl environment
```

如果想更改这个默认的配置，我们可以在/etc/rabbitmq/目录下创建一个rabbitmq.config文件，配置信息可以按照指定的json规则进行指定。如下所示：

```
1  [{
2    rabbit,
3
4    [
5
6    {
7      queue_index_embed_msgs_below,
8      4097
9    }
10   ]
11  }].
```

然后重启rabbitmq服务（rabbitmqctl stop----> rabbitmq-server -detached）。

那么我们是不是把queue_index_embed_msgs_below参数的值调节的越大越好呢？肯定不是的

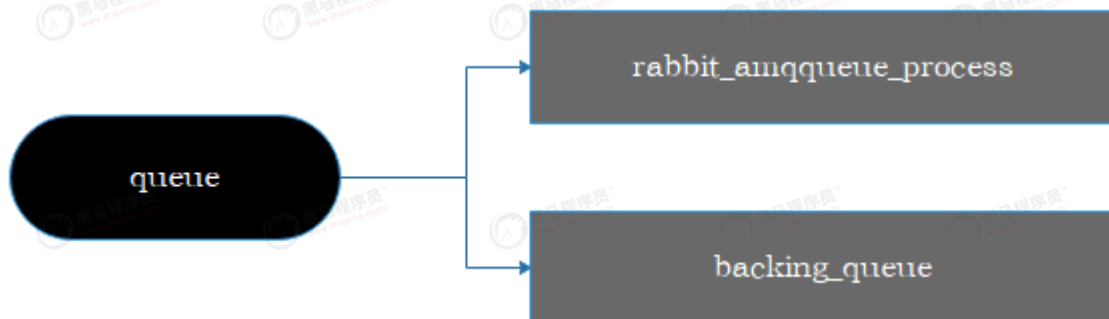
rabbit_queue_index中以顺序（文件名从0开始累加）的段文件来进行存储，后缀为".idx",每个段文件中包含固定的SEGMENT_ENTRY_COUNT条记录，SEGMENT_ENTRY_COUNT默认值为

16384。每个rabbit_queue_index从磁盘中读取消息的时候至少在内存中维护一个段文件，所以设置queue_index_embed_msgs_below值的时候需要格外谨慎，一点点增大也可能会引起内存爆炸式增长。

相关知识：队列的结构

队列的结构以及消息的状态

Rabbitmq中队列的是由两部分组成：rabbit_amqpqueue_process和backing_queue组成：



rabbit_amqpqueue_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_amqpqueue_process提供相关的接口以供调用。

如果消息发送的队列是空的且队列有消费者，该消息不会经过该队列而是直接发往消费者，如果无法直接被消费，则需要将消息暂存入队列，以便重新投递。消息在存入队列后，

主要有以下几种状态：

- alpha: 消息内容(包括消息体、属性和headers)和消息索引都存在内存中（消耗内存最多，CPU消耗最少）
- beta: 消息内容保存在磁盘中，消息索引都存在内存中（只需要一次IO操作就可以读取到消息）
- gamma: 消息内容保存在磁盘中，消息索引在磁盘和内存中都存在（只需要一次IO操作就可以读取到消息）
- delta: 消息内容和消息索引都在磁盘中（消耗内存最小，但是会消耗更多的CPU和磁盘的IO操作）

持久化的消息，消息内容和消息索引必须先保存在磁盘中，才会处于上面状态中的一种，gamma状态只有持久化的消息才有这种状态。Rabbitmq在运行时会根据统计的消息传送速度

定期计算一个当前内存中能够保存的最大消息数量 (target_ram_count) ,如果alpha状态的消息数量大于此值时,就会引起消息的状态转换,多余的消息可能会转换到beta状态、

gamma状态或者delta状态。区分这4种状态的主要作用是满足不同的内存和CPU 的需求。

对于普通队列而言, backing_queue内部的实现是通过5个子队列来体现消息的状态的:

- Q1: 只包含alpha状态的消息
- Q2: 包含beta和gamma的消息
- Delta: 包含delta的消息
- Q3: 包含beta和gamma的消息
- Q4: 只包含alpha状态的消息



一般情况下, 消息按照Q1->Q2->Delta->Q3->Q4这样的顺序进行流动, 但并不是每一条消息都会经历所有状态, 这取决于当前系统的负载情况 (比如非持久化的消息在内存负载不高时,

就不会经历delta)。如此设计的好处: 可以在队列负载很高的情况下, 能够通过将一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,

使得整个队列具有良好的弹性。

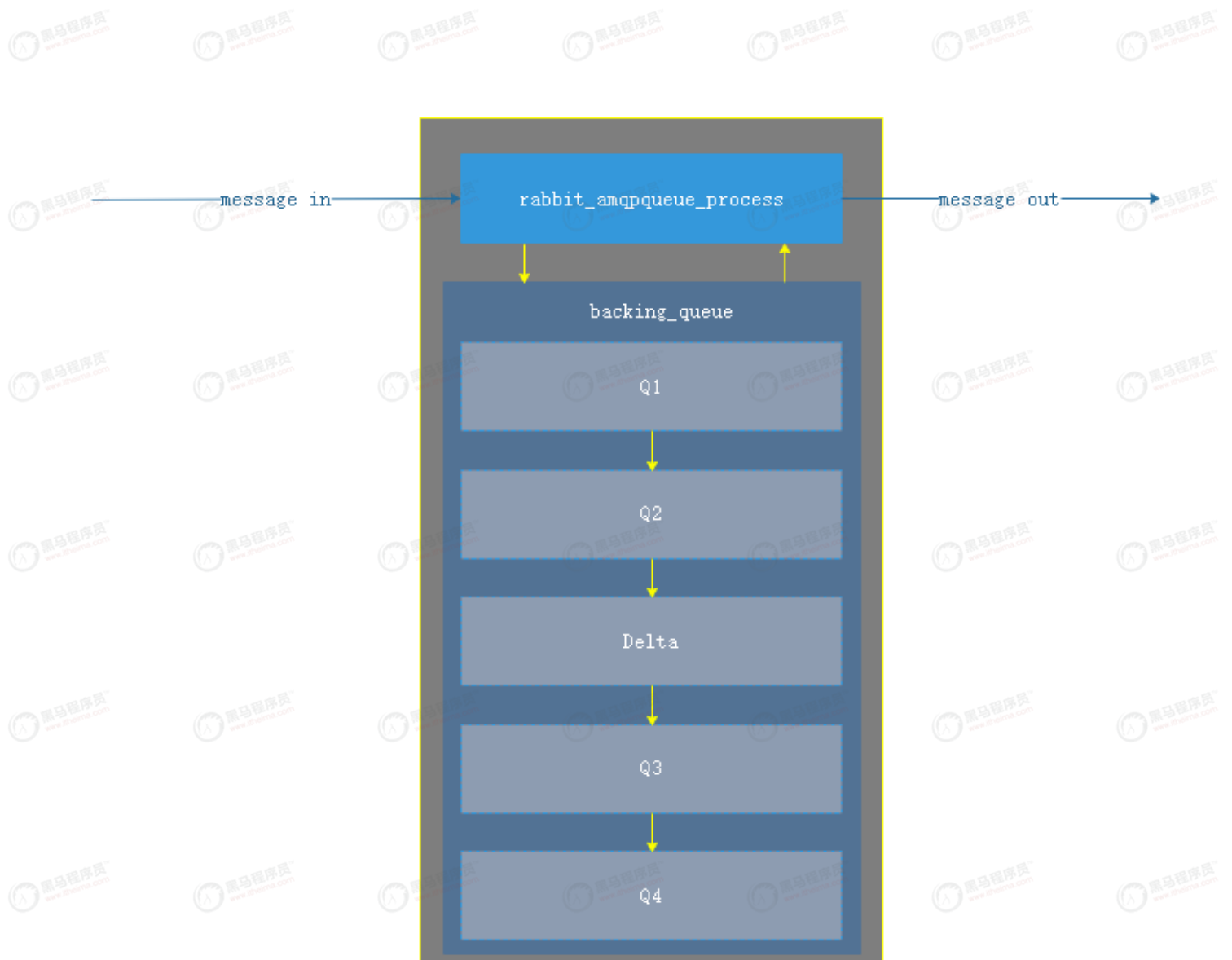
消费消息时的状态转换

消费者消费消息也会引起消息状态的转换，状态转换的过程如下所示：

1. 消费者消费时先从Q4获取消息，如果获取成功则返回。
2. 如果Q4为空，则从Q3中获取消息，首先判断Q3是否为空，如果为空返回队列为空，即此时队列中无消息
3. 如果Q3不为空，取出Q3的消息，然后判断Q3和Delta中的长度，如果都为空，那么Q2、Delta、Q3、Q4都为空，直接将Q1中的消息转移至Q4，下次直接从Q4中读取消息
4. 如果Q3为空，Delta不为空，则将Delta中的消息转移至Q3中，下次直接从Q3中读取。
5. 在将消息从Delta转移至Q3的过程中，是按照索引分段读取，首先读取某一段，然后判断读取的消息个数和Delta消息的个数，如果相等，判定Delta已无消息，直接将读取 Q2和读取到消息一併放入Q3，如果不相等，仅将此次读取的消息转移到Q3。

通常在负载正常时，如果消息被消费的速度不小于接收新消息的速度，对于不需要保证可靠性的消息来说，极有可能只会处于alpha状态。对于durable属性设置为true的消息，它一定会进入

gamma状态，并且在开启publisher confirm机制时，只有到了gamma状态时才会确认该消息已被接收，若消息消费速度足够快、内存也充足，这些消息也不会继续走到下一个状态。



在系统负载较高中，已经收到的消息若不能很快被消费掉，就是这些消息就是在队列中"堆积"，那么此时Rabbitmq就需要花更多的时间和资源处理"堆积"的消息，如此用来处理新流入的消息的能

力就会降低，使得流入的消息又被"堆积"继续增大处理每个消息的平均开销，继而情况变得越来越恶化，使得系统的处理能力大大降低。

减少消息堆积的常见解决方案：

- 1、增加prefetch_count的值，设置消费者存储未确认的消息的最大值
- 2、消费者进行multiple ack，降低ack带来的开销

相关知识：惰性队列

默认情况下，当生产者将消息发送到Rabbitmq的时候，队列中的消息会尽可能地存储在内存中，这样可以更快地将消息发送给消费者。即使是持久化的消息，在被写入磁盘的同时也会

在内存中驻留一份备份。这样的机制无形会占用更多系统资源，毕竟内存应该留给更多有需要的地方。如果发送端过快或消费端宕机，导致消息大量积压，此时消息还是在内存和磁盘各

存储一份，在消息大爆发的时候，MQ服务器会撑不住，影响其他队列的消息收发，能不能有效的处理这种情况呢。答案 **惰性队列**。

RabbitMQ从3.6.0版本开始引入了惰性队列（Lazy Queue）的概念。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”。注意如果惰性队列中存储的是非持久化的消息，内存的使用率会

一直很稳定，但是重启之后消息一样会丢失。

把一个队列设置成惰性队列的方式：

```
1 // 声明队列
2 @Bean(name = "direct.queue_03")
3 public Queue commonQueue03() {
4     QueueBuilder queueBuilder = QueueBuilder.durable("direct.queue_03");
5     queueBuilder.lazy(); // 把队列设置成惰性队列
6     return queueBuilder.build();
7 }
```

2.3.2 第二种情况

场景描述

消费者消费到这个消息但是还没有及时处理，消费者宕机了。

解决方案

默认情况下，消费者消费到这个消息以后会自动给服务端发送一个Basic.Ack指令，告知这个消息已经被消费了，此时服务端会将这个消息从内存（磁盘）删除掉。

针对上述消息丢失的场景：我们只需要将自动应答更改为手动应答即可。

具体的实现如下所示：

1、更改消费端应答模式为：手动应答

```
1 listener:
2     simple:
3     acknowledge-mode: manual # 更改消息的应答模式为手动应答
```

2、当消费者消费完消息以后，进行消息应答

```
1 @RabbitListener(queues = "direct.queue_01")
2 public void directExchangeQueue01(String messageBody , Channel channel
3 , Message message) {
4     try {
5         // 对消息进行消费
6         LOGGER.info("direct exchange queue 01 message is : {}" ,
7 messageBody);
8
9         // 进行手动应答,第二个参数表示是否需要将该消息之前的所有的消息都进行应答
10
11         channel.basicAck(message.getMessageProperties().getDeliveryTag() ,
12 false);
13
14     } catch (IOException e) {
15         e.printStackTrace();
16         LOGGER.error("direct exchange queue 01 处理失败, message is :
17 {}" , messageBody);
18     }
19 }
```

相关知识：消息分发机制

默认的分发机制：当Rabbitmq队列拥有多个消费者时，队列收到的消息将以轮询的方式发给消费者。每条消息只会发给订阅列表里的一个消费者。这种方式非常适合扩展，而且它是

专门为并发程序设计的。如果现在负载加重，那么只需要创建更多的消费者来消费处理消息即可。

很多时候轮询的分发机制也不是那么的优雅。默认情况下，如果有n个消费者，那么Rabbitmq会将第m条消息分发给第 $m\%n$ （取余的方式）个消费者，Rabbitmq不管消费者是否消费

并已经确认了消息。试想一下，如果某些消费者任务繁重，来不及消费那么多的消息，而某些其他的消费者由于某种原因（比如业务逻辑简单、机器性能卓越等）很快地处理完了分配

到的消息，进而进程空闲，这样就会造成整体应用吞吐量的下降。

那么该如何处理呢？这里就要用到`channel.basicQos(int prefetchCount)`这个方法。`channel.basicQos`方法允许限制通道上的消费者所能够保持的最大未确认消息的数量。

```
1 | spring.rabbitmq.listener.simple.prefetch: 1 # 设置队列中最大的未确认的消息数量
```

2.3.3 第三种情况

场景描述

生产者将消息发送给交换机以后，正当交换机将这个信息发送给指定队列的时候，该队列所在的主机宕机了，那么这一则消息就会丢失。

解决方案

要想解决这种情况下消息的丢失，我们就需要知道生产者针对该消息的投递结果。默认情况下发送消息的操作，服务端是不会返回任何信息给生产者的，也就是说默认情况下生产者是

不知道消息有没有正确地到达服务器端。那么要想知道生产者针对该消息的投递结果，我们有两种解决方案：

- 1、通过事务机制实现
- 2、通过发送方确认（publisher confirm）机制实现

事务机制的实现方案如下所示：

在配置类中配置Rabbitmq的事务管理器：

```
1 // 配置事务管理器
2 @Bean(name = "rabbitTransactionManager")
3 public RabbitTransactionManager
4     rabbitTransactionManager(ConnectionFactory connectionFactory) {
5     return new RabbitTransactionManager(connectionFactory) ;
6 }
```

定义发送消息的类：

```
1 @Component
2 public class RabbitmqProducer {
3
4     // 定义日志记录器
5     private static final Logger LOGGER =
6     LoggerFactory.getLogger(RabbitmqProducer.class) ;
7
8     @Autowired
9     private RabbitTemplate rabbitTemplate ;
10
11     @Transactional(rollbackFor = Exception.class , transactionManager =
12     "rabbitTransactionManager")
13     public void sendMessage() {
14
15         rabbitTemplate.setChannelTransacted(true); // 将消息通道设置
16         // 为事务机制
17         String msg = "测试生产者事务消息" ;
18         rabbitTemplate.convertAndSend("direct.exchange" , "create" ,
19         msg.getBytes());
20         int a = 1 / 0 ;
21         LOGGER.info("transactionManager message send success ----> " +
22         msg);
23     }
24 }
```

在调用方进行try...catch处理（可以重新尝试发送消息）

```

1 // 测试事务消息
2 private static void
  sendTransactionManagerMsg(ConfigurableApplicationContext
    applicationContext) {
3
4     RabbitmqProducer rabbitmqProducer =
      applicationContext.getBean(RabbitmqProducer.class);
5     try {
6         rabbitmqProducer.sendMessage();
7     } catch (Exception e) {
8         e.printStackTrace();
9         System.out.println("事务消息回滚了");
10    }
11
12 }

```

我们也可以通过Wireshark捕获到命令的传递过程，如下所示：

211	119.347987	192.168.23.1	192.168.23.131	AMQP	67 Channel.Open
212	119.350269	192.168.23.131	192.168.23.1	AMQP	70 Channel.Open-Ok
213	119.350721	192.168.23.1	192.168.23.131	AMQP	66 Tx.Select
214	119.351363	192.168.23.131	192.168.23.1	AMQP	66 Tx.Select-Ok
215	119.378609	192.168.23.1	192.168.23.131	AMQP	180 Basic.Publish x=direct.exchange rk=create Content-Header type=application/octet-stream Content-Body
216	119.380182	192.168.23.1	192.168.23.131	AMQP	66 Tx.Rollback
218	119.383339	192.168.23.131	192.168.23.1	AMQP	66 Tx.Rollback-Ok

事务机制会影响Rabbitmq的性能，事务机制在一条消息发送之后会使发送端阻塞，以等待RabbitmqMQ的回应，之后才能继续发送下一条消息。因此在真实开发过程中很少的使用。

Rabbitmq提供了一个改进方案，即发送方确认机制（publisher confirm）。

发送方确认（publisher confirm）机制原理说明：

生产者将信道设置成confirm（确认）模式，一旦信道进入confirm模式，所有在该信道上发布消息都会被指派一个唯一的ID（从1开始），一旦消息被投递到所有匹配的队列之后，

Rabbitmq就会发送一个确认(Basic.ACK)给生产者（包含消息的唯一ID），这就使得生产者知晓消息已经正确到达目的地了。如果消息和队列是可持久化的，那么确认消息会在消息写

入磁盘之后发出。

与事务机制相比，发送方确认机制最大的好处在于它是**异步**的，一旦发布一条消息，生产者应用程序就可以在等信道返回确认的同时继续发送下一条消息，当消息最终得到确认之后，

生产者应用程序便可以通过回调方法来处理该确认消息。如果RabbitMQ因为自身内部错误导致消息丢失，就会发送一条Basic.Nack命令，生产者应用程序同样可以在回调方法中处理

该nack命令。

发送方确认（publisher confirm）机制实现：

1、在配置文件中配置开启生产者确认机制

```
1 | spring.rabbitmq.publisher-confirm-type: correlated # 开启生产者确认机制
```

2、定义发送消息的类：

```
1 | @Component
2 | public class RabbitmqSendMsgConfirm implements
   | RabbitTemplate.ConfirmCallback {
3 |
4 |     // 定义日志记录对象
5 |     private static final Logger LOGGER =
   | LoggerFactory.getLogger(RabbitmqSendMsgConfirm.class) ;
6 |
7 |     @Autowired
8 |     private RabbitTemplate rabbitTemplate ;
9 |
10 |    // 发送消息
11 |    public void sendMsg(String msg , String exchangeName , String
   | routingKey) {
12 |
13 |        // 设置消息发送是否成功的回调
14 |        rabbitTemplate.setConfirmCallback(this);
15 |
16 |        // void convertAndSend(String exchange, String routingKey,
   | Object message, MessagePostProcessor messagePostProcessor,
   | CorrelationData correlationData)
17 |        CorrelationData correlationData = new CorrelationData(msg) ;
```

```

18         rabbitTemplate.convertAndSend(exchangeName , routingKey , msg ,
message -> {
19
20             message.getMessageProperties().setDeliveryMode(MessageDeliveryMode.PER
SISTENT);           // 进行消息持久化操作
21             return message ;
22         } , correlationData);
23     }
24
25     /**
26      * CorrelationData: 每个发送的消息都需要配备一个 CorrelationData 相关数据
对象, CorrelationData 对象内部只有一个 id 属性, 用来表示当前消息唯一性。
27      * @param correlationData
28      * @param ack
29      * @param cause
30      */
31     @Override
32     public void confirm(CorrelationData correlationData, boolean ack,
String cause) {
33         if(ack) {
34             LOGGER.info("消息发送成功");
35         }else {
36             LOGGER.info("消息发送失败");
37         }
38     }
39
40 }

```

上述这种确认机制，是确保消息是否已经发送到正确的交换机上。

消息一旦被发送到正确到交换机上以后，ack的值就是true。那么这个消息是否被投递到了指定的队列，如果消息没有被投递到指定的队列。那么作为生产者如何获知到呢？

我们需要在次进行确认，具体的实现如下所示：

1、在配置文件中进行如下配置

```

1  spring.rabbitmq.template.mandatory: true    (表示的意思是交换机无法根据自身的类
型和路由键找到一个符号条件的队列，那么RabbitmqMQ会调用Basic.Return返回消息给生产
者)
2  spring.rabbitmq.publisher-returns: true      开启return确认机制

```

2、添加return确认机制的回调函数

```
1 // 发送消息
2 public void sendMsg(String msg , String exchangeName , String
  routingKey) {
3
4     ...
5     // 添加return确认机制的回调函数
6     rabbitTemplate.setReturnCallback(this);
7     ...
8
9 }
10
11 // 当消息没有被交换机投递到指定的队列的回调函数
12 @Override
13 public void returnedMessage(Message message, int replyCode, String
  replyText, String exchange, String routingKey) {
14     LOGGER.info("消息没有被投递到指定的队列 ---> " + new
  String(message.getBody()));
15 }
```

2.4 RabbitMQ秒杀公平性保证

消息的可靠性传输可以保证秒杀业务的公平性。关于秒杀业务的公平性，我们还需要考虑一点：消息的顺序性（先进入队列的消息先进行处理）

2.4.1 RabbitMQ消息顺序性说明

顺序性：消息的顺序性是指消费者消费到的消息和发送者发布的消息的顺序是一致的。

举个例子，不考虑消息重复的情况，如果生产者发布的消息分别为msg1、msg2、msg3，那么消费者必然是按照msg1、msg2、msg3的顺序进行消费的。

目前很多资料显示RabbitMQ的消息能够保障顺序性，这是不正确的，或者说这个观点有很大的局限性。在不使用任何RabbitMQ的高级特性，也没有消息丢失、网络故障之类异常的

情况发生，并且只有一个消费者的情况下，也只有一个生产者的情况下可以保证消息的顺序性。如果有多个生产者同时发送消息，无法确定消息到达Broker 的前后顺序，也就无法验

证消息的顺序性，因为每一次消息的发送都是在各自的线程中进行的。

2.4.2 RabbitMQ消息顺序错乱演示

生产者发送消息：

- 1、不使用生产者确认机制，单生产者单消费者可以保证消息的顺序性
- 2、使用了生产者确认机制，那么就无法保证消息到达Broker的前后顺序，因为消息的发送是异步发送的，每一个线程的执行时间不同
- 3、生产端使用事务机制，保证消息的顺序性

消费端消费消息：

- 1、单消费者可以保证消息的顺序性
- 2、多消费者不能保证消息的顺序，因为每一个消息的消费都是在各自的线程中进行，每一个线程的执行时间不同

2.4.3 RabbitMQ消息顺序性保障

生产端启用事务机制，单生产者单消费者。如果我们不考虑消息到达MQ的顺序，只是考虑对已经到达MQ中的消息顺序消费，那么需要保证消费者是单消费者即可。

2.5 RabbitMQ秒杀不超卖保证

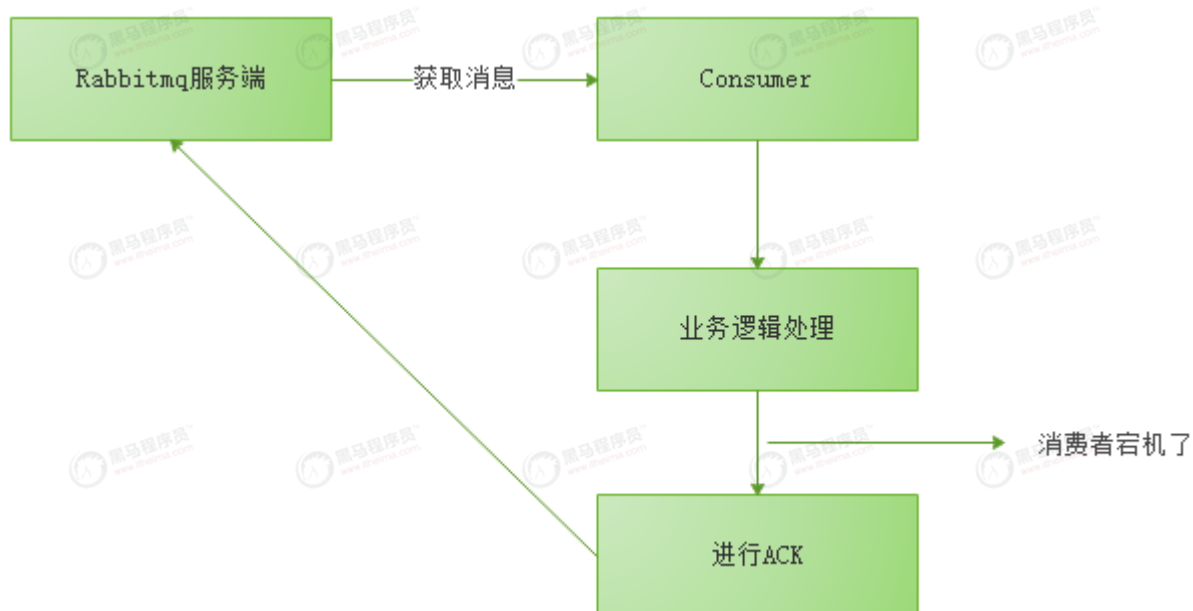
要保证秒杀不超卖，我们需要在很多的环节进行考虑。比如：在进行预扣库存的时候，我们需要考虑不能超卖，在进行数据库真实库存扣减的时候也需要考虑不能超卖。而对我们的mq

这个环节而言，要保证不超卖我们只需要保证消息不被重复消费。

2.5.1 重复消费原因说明

首先我们可以确认的是，触发消息重复执行的条件会是很苛刻的！也就说在大多数场景下不会触发该条件！！！一般出在任务超时，或者没有及时返回状态，引起任务重新入队列，

重新消费！在rabbitmq里连接的断开也会触发消息重新入队列。如下图所示：



由于服务端没有收到消费端的ack应答，因此该条消息还会重新进行投递。

2.5.2 幂等性保障方案

重复消费不可怕，可怕的是你没考虑到重复消费之后，**怎么保证幂等性**。

所谓幂等性，就是对接口的多次调用所产生的结果和调用一次是一致的。通俗点说，就一个数据，或者一个请求，给你重复来多次，你得确保对应的数据是不会改变的，**不能出错**。

举个例子：

假设你有个系统，消费一条消息就往数据库里插入一条数据，要是你一条消息重复消费两次，你不就插入了两条，这数据不就错了？但是你要是消费到第二次的时候，自己判断一下是否

已经消费过了，若是就直接扔了，这样不就保留了一条数据，从而保证了数据的正确性。一条数据重复消费两次，数据库里就只有一条数据，这就保证了系统的幂等性。

怎么保证消息队列消费的幂等性？这一点需要结合的实际的业务来进行处理：

1、比如我们消费的数据需要写数据库，你先根据主键查一下，如果这数据都有了，你就别插入了，执行以下update操作

2、比如我们消费的数据需要写Redis，那没问题了，反正每次都是 set，天然幂等性。

3、比如你不是上面两个场景，那做的稍微复杂一点，你需要让生产者发送每条数据的时候，里面加一个**全局唯一的 id**，类似订单 id 之类的东西，然后你这里消费到了之后，先根据这个

id 去比如 Redis 里查一下，之前消费过吗？如果没有消费过，你就处理，然后这个 id 写 Redis。如果消费过了，那你就别处理了，保证别重复处理相同的消息即可。

4、比如基于数据库的唯一键来保证重复数据不会重复插入多条。因为有唯一键约束了，重复数据插入只会报错，不会导致数据库中出现脏数据。

2.6 RabbitMQ高可用

Rabbitmq常见的部署模式：单机，普通集群，镜像集群

2.6.1 RabbitMQ普通集群

集群原理

思考的问题：

1、搭建集群的好处？

- 提供整体消息队列服务的可靠性
- 可以通过水平扩容提高整体服务的吞吐量

2、有了集群以后是否可以保证消息不丢失？

- 不可以

基于存储空间和性能的考虑，在Rabbitmq集群中创建队列，集群**只会在单个节点而不是在所有节点上创建队列的进程**并包含完整的队列消息（元数据，状态，内容）。这样只有队列的

宿主节点，即所有者节点知道队列的所有信息，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那个节点的指针。因此当集群节点崩溃时，该节点的队列进程和关联的绑定

都会消失。附加在哪些队列上的消费者也会丢失其所订阅的消息，并且任何匹配该队列绑定信息的新消息也都会消失。

3、集群中的交换机有哪些特点呢？

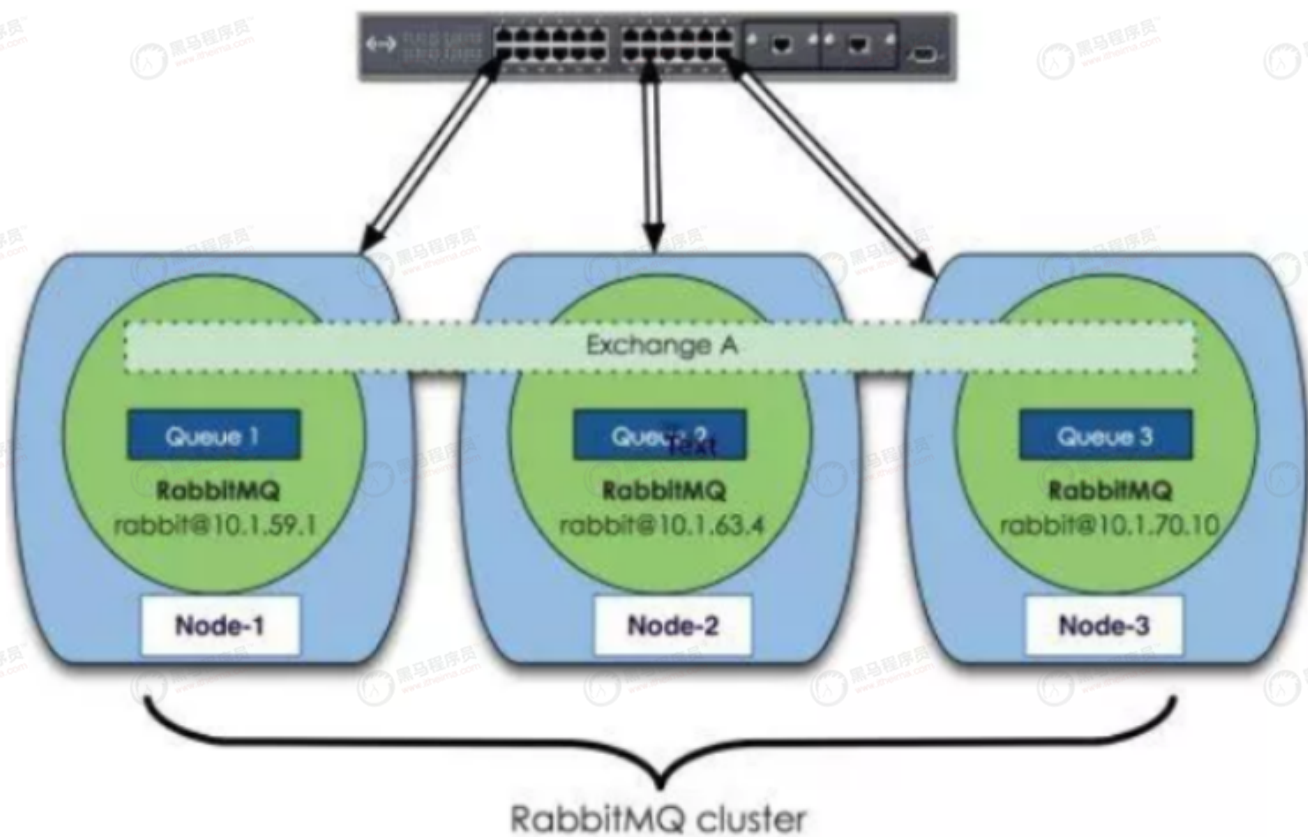
- 交换机其实只是一个名称和绑定列表，没有自己独立的进程，并且交换机的元数据信息会在多个rabbitmq节点上进行共享

当消息发布到交换机时，实际上是由所连接信道将消息上的路由键同交换机的绑定列表进行比较，然后再路由消息。当创建一个新的交换机时，Rabbitmq所要做的就是将绑定列表添加

到集群中的所有节点上，这样每个节点上的每条信道都可以访问到新的交互机了。

4、客户端与集群建立连接的时候，是否需要与集群中所有的节点建立连接？

- 只需要连接集群中任意一个节点就可以了



集群搭建

创建容器

接下来我们就来搭建一个rabbitmq的集群，本次我们搭建一个具有3个节点的rabbitmq集群。

1、拉取镜像

```
1 | docker pull rabbitmq:3.6.10-management
```

2、创建容器

```
1 | docker run -di --network=docker-network --ip=172.19.0.50 --  
hostname=rabbitmq-node01 --name=rabbitmq_01 -p 15673:15672 -p 5673:5672  
--privileged=true -e RABBITMQ_ERLANG_COOKIE='rabbitcookie'  
rabbitmq:3.6.10-management /bin/bash  
2 | docker run -di --network=docker-network --ip=172.19.0.51 --  
hostname=rabbitmq-node02 --name=rabbitmq_02 -p 15674:15672 -p 5674:5672  
--privileged=true -e RABBITMQ_ERLANG_COOKIE='rabbitcookie'  
rabbitmq:3.6.10-management /bin/bash  
3 | docker run -di --network=docker-network --ip=172.19.0.52 --  
hostname=rabbitmq-node03 --name=rabbitmq_03 -p 15675:15672 -p 5675:5672  
--privileged=true -e RABBITMQ_ERLANG_COOKIE='rabbitcookie'  
rabbitmq:3.6.10-management /bin/bash
```

参数说明：Erlang Cookie值必须相同，也就是RABBITMQ_ERLANG_COOKIE参数的值必须相同。因为RabbitMQ是用Erlang实现的，Erlang Cookie相当于不同节点之间相互通讯的密钥，

Erlang节点通过交换Erlang Cookie获得认证。

3、进入到rabbitmq的容器中

```
1 | docker exec -it rabbitmq_01 /bin/bash
```

4、配置hosts文件，让各个节点都能互相识别对方的存在。在系统中编辑 /etc/hosts文件，添加ip地址和节点名称的映射信息(apt-get update , apt-get install vim):

```
1 172.19.0.50 rabbitmq-node01
2 172.19.0.51 rabbitmq-node02
3 172.19.0.52 rabbitmq-node03
```

5、启动rabbitmq，并且查看状态

```
1 root@014faa4cba72:/# rabbitmq-server -detached # 启动
  rabbitmq服务，该命令可以启动erlang虚拟机和rabbitmq服务
2 root@014faa4cba72:/# rabbitmqctl status # 查看节点
  信息
3 Status of node rabbit@014faa4cba72
4 [{pid,270},
5  {running_applications,
6    [{rabbitmq_management,"RabbitMQ Management Console","3.6.10"},
7     {rabbitmq_management_agent,"RabbitMQ Management Agent","3.6.10"},
8     {rabbitmq_web_dispatch,"RabbitMQ Web Dispatcher","3.6.10"},
9     .....
10 root@014faa4cba72:/# rabbitmqctl cluster_status # 查看集群
   节点状态
11 Cluster status of node rabbit@014faa4cba72
12 [{nodes,[{disc,[rabbit@014faa4cba72]}]},
13  {running_nodes,[rabbit@014faa4cba72]}, # 正在运行
   的只有一个节点
14  {cluster_name,<<"rabbit@014faa4cba72">>},
15  {partitions,[]},
16  {alarms,[{rabbit@014faa4cba72,[]}]}
```

注意：此时我们可以通过浏览器访问rabbitmq的后端管理系统，但是rabbitmq默认提供的guest用户不支持远程访问。因此我们需要创建用户，并且对其进行授权

```

1 root@014faa4cba72:/# rabbitmqctl add_user admin admin # 添加用户，用户名
  为admin，密码为admin
2 root@014faa4cba72:/# rabbitmqctl list_users # 查看rabbitmq的
  用户列表
3 Listing users
4 admin [] # admin用户已经添
  加成功，但是没有角色
5 guest [administrator]
6 root@014faa4cba72:/# rabbitmqctl set_user_tags admin administrator #
  给admin用户设置管理员权限
7 # rabbitmqctl delete_user admin # 删除admin用户
8 # rabbitmqctl stop_app # 停止rabbitmq服务
9 # rabbitmqctl stop # 会将rabbitmq的服务和erlang虚拟机一同关闭

```

再次使用admin用户就可以登录web管理系统了。在其他的rabbitmq中也创建用户，以便后期可以访问后端管理系统。

为admin用户设置可以访问的虚拟机:

RabbitMQ

Overview Connections Channels Exchanges Queues **Admin**

Cluster: rabbitmq-node02 (change) User: admin Log out
RabbitMQ 3.6.10, Erlang 19.2.1

Virtual Hosts

Users Policies

1 item, page size up to 100

Overview		Messages		Network		Message rates	
Name	Users (?)	Ready	Unacked	Total	From client	To client	publish deliver / get
/	admin, guest	0	0	0	0B/s	0B/s	0.00/s 0.00/s

Filter: Regexp (?) (?)

▼ Add a new virtual host

Name:

Add virtual host

配置集群

1、同步cookie

集群中的Rabbitmq节点需要通过交换密钥令牌以获得相互认证，如果节点的密钥令牌不一致，那么在配置节点时就会报错。

获取某一个节点上的/var/lib/rabbitmq/.erlang.cookie文件，然后将其复制到其他的节点上。我们以node01节点为基准，进行此操作。

```
1 docker cp rabbitmq_01:/var/lib/rabbitmq/.erlang.cookie .
2 docker cp .erlang.cookie rabbitmq_02:/var/lib/rabbitmq
3 docker cp .erlang.cookie rabbitmq_03:/var/lib/rabbitmq
```

2、建立集群关系

目前3个节点都是独立的运行，之间并没有任何的关联关系。接下来我们就来建立3者之间的关联关系，我们以rabbitmq-node01为基准，将其他的两个节点加入进来。

把rabbitmq-node02加入到节点1中

```
1 # 进入到rabbitmq-node02中
2 rabbitmqctl stop_app # 关闭rabbitmq服务
3 rabbitmqctl reset    # 进行重置
4 rabbitmqctl join_cluster rabbit@rabbitmq-node01 # rabbitmq-node01为
   节点1的主机名称
5 rabbitmqctl start_app # 启动rabbitmq节点
```

把rabbitmq-node03加入到节点1中

```
1 # 进入到rabbitmq-node03中
2 rabbitmqctl stop_app # 关闭rabbitmq服务
3 rabbitmqctl reset    # 清空节点的状态，并将其恢复都空白状态，当设置的节点时集群
   中的一部分，该命令也会和集群中的磁盘节点进行通讯，告诉他们该节点正在离开集群。不然集群
   会认为该节点处理故障，并期望其最终能够恢复过来
4 rabbitmqctl join_cluster rabbit@rabbitmq-node01 # rabbitmq-node01为
   节点1的主机名称
5 rabbitmqctl start_app # 启动rabbitmq节点
```

进入后台管理系统查看集群概述：

Global counts (?)

Connections: 0

Channels: 0

Exchanges: 10

Queues: 10

Consumers: 0

Nodes

Name	File descriptors (?)	Socket descriptors (?)	Erlang processes	Memory	Disk space	Roles mode	Info	Reset stats DB	+/-
rabbit@rabbitmq-node01	35 1048576 available	0 943626 available	352 1048576 available	78MB 13GB high watermark48MB low watermark	40GB	basic	Disc 1	Reset	
rabbit@rabbitmq-node02	25 1048576 available	0 943626 available	327 1048576 available	72MB 13GB high watermark48MB low watermark	40GB	basic	Disc 1	Reset	
rabbit@rabbitmq-node03	24 1048576 available	0 943626 available	327 1048576 available	70MB 13GB high watermark48MB low watermark	40GB	basic	RAM 1	Reset	

Reset stats on all nodes

至此也就证明我们的rabbitmq集群就已经搭建好了。

节点类型

节点类型介绍

在使用**rabbitmqctl cluster_status**命令来查看集群状态时会有[{nodes,[{disc,['rabbit@rabbitmq-node01','rabbit@rabbitmq-node02','rabbit@rabbitmq-node03']}]}这一项信息，

其中的disc标注了Rabbitmq节点类型。Rabbitmq中的每一个节点，不管是单一节点系统或者是集群中的一部分要么是内存节点，要么是磁盘节点。内存节点将所有的队列，交换

机，绑定关系、用户、权限和vhost的元数据定义都存储在内存中，而磁盘节点则将这些信息存储到磁盘中。单节点的集群中必然只有磁盘类型的节点，否则当重启Rabbitmq之后

，所有关于系统配置信息都会丢失。不过在集群中，可以选择配置部分节点为内存节点，这样可以获得更高的性能。

节点类型变更

如果我们没有指定节点类型，那么默认就是磁盘节点。我们在添加节点的时候，可以使用如下的命令来指定节点的类型为内存节点：

```
1 | rabbitmqctl join_cluster rabbit@rabbitmq-node01 --ram
```

我们也可以使用如下的命令将某一个磁盘节点设置为内存节点：

```
1 | rabbitmqctl change_cluster_node_type {disc , ram}
```

如下所示：

```
1 root@rabbitmq-node02:/# rabbitmqctl stop_app
  # 关闭rabbitmq服务
2 Stopping rabbit application on node 'rabbit@rabbitmq-node02'
3 root@rabbitmq-node02:/# rabbitmqctl change_cluster_node_type ram
  # 将root@rabbitmq-node02节点类型切换为内存节点
4 Turning 'rabbit@rabbitmq-node02' into a ram node
5 root@rabbitmq-node02:/# rabbitmqctl start_app
  # 启动rabbitmq服务
6 Starting node 'rabbit@rabbitmq-node02'
7 root@rabbitmq-node02:/# rabbitmqctl cluster_status
  # 查看集群状态
8 Cluster status of node 'rabbit@rabbitmq-node02'
9 [{nodes,[{disc,['rabbit@rabbitmq-node03','rabbit@rabbitmq-node01']},
10 {ram,['rabbit@rabbitmq-node02']}]},
11 {running_nodes,['rabbit@rabbitmq-node01','rabbit@rabbitmq-node03',
12 'rabbit@rabbitmq-node02']},
13 {cluster_name,<<"rabbit@rabbitmq-node01">>},
14 {partitions,[],},
15 {alarms,[{'rabbit@rabbitmq-node01',[]},
16 {'rabbit@rabbitmq-node03',[]},
17 {'rabbit@rabbitmq-node02',[]}]}}
18 root@rabbitmq-node02:/#
```

节点选择

Rabbitmq只要求在集群中至少有一个磁盘节点，其他所有的节点可以是内存节点。当节点加入或者离开集群时，它们必须将变更通知到至少一个磁盘节点。如果只有一个磁盘节点，而且

不凑巧它刚好崩溃了，那么集群可以继续接收和发送消息。但是不能执行创建队列，交换机，绑定关系、用户已经更改权限、添加和删除集群节点操作了。也就是说、如果集群中唯一的磁

盘节点崩溃了，集群仍然可以保持运行，但是知道将该节点恢复到集群前，你无法更改任何东西，所以在创建集群的时候应该保证至少有两个或者多个磁盘节点。当内存节点重启后，它会

连接到预先配置的磁盘节点，下载当前集群元数据的副本。当在集群中添加内存节点的时候，确保告知所有的磁盘节点（内存节点唯一存储到磁盘中的元数据信息是磁盘节点的地址）。只

要内存节点可以找到集群中至少一个磁盘节点，那么它就能在重启后重新加入集群中。

集群测试

连接集群中某一个节点进行消息的收发操作。

集群优化

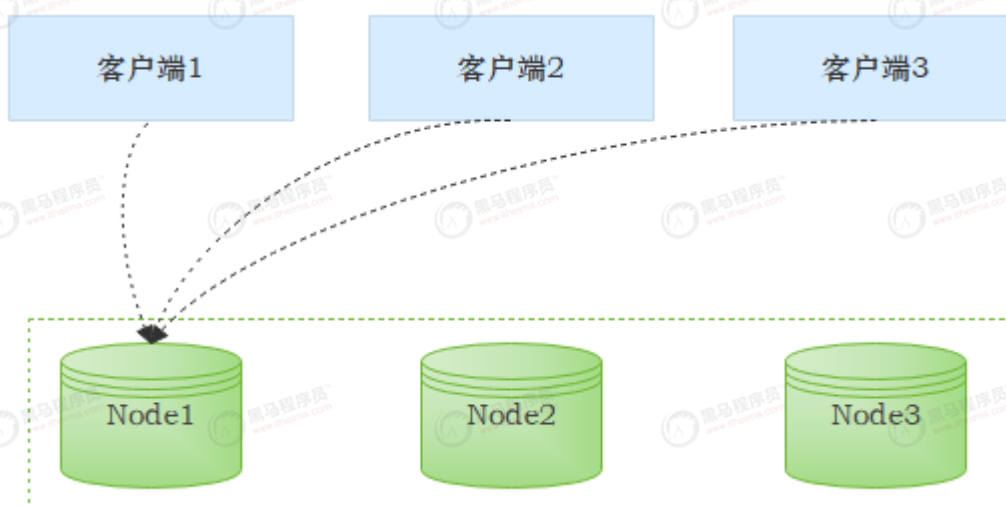
之前搭建的集群存在的问题：不具有负载均衡能力

优化架构

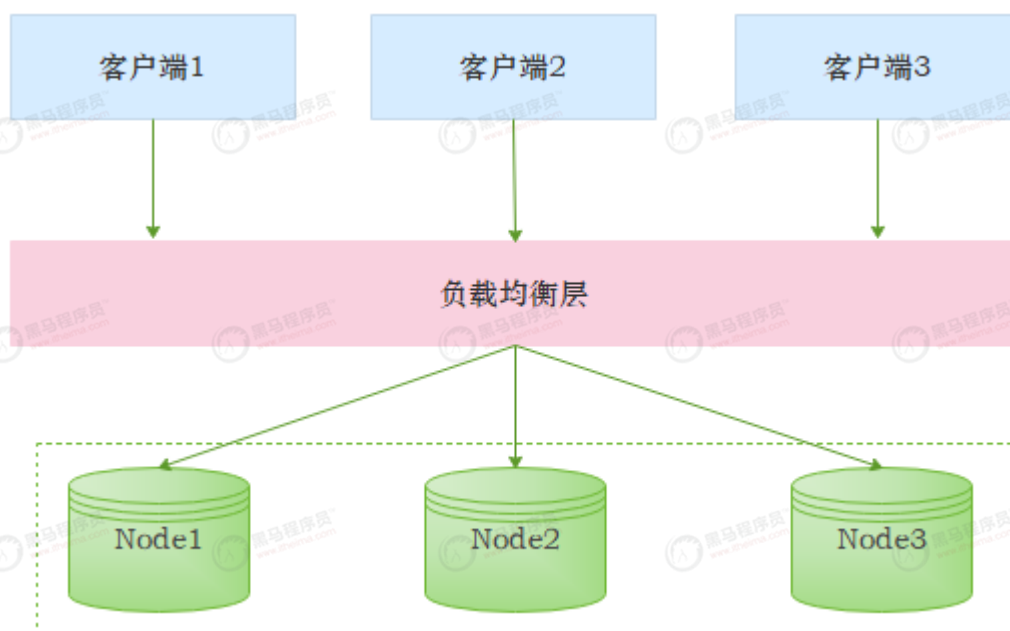
思考的问题：

1、为什么要进行负载均衡？

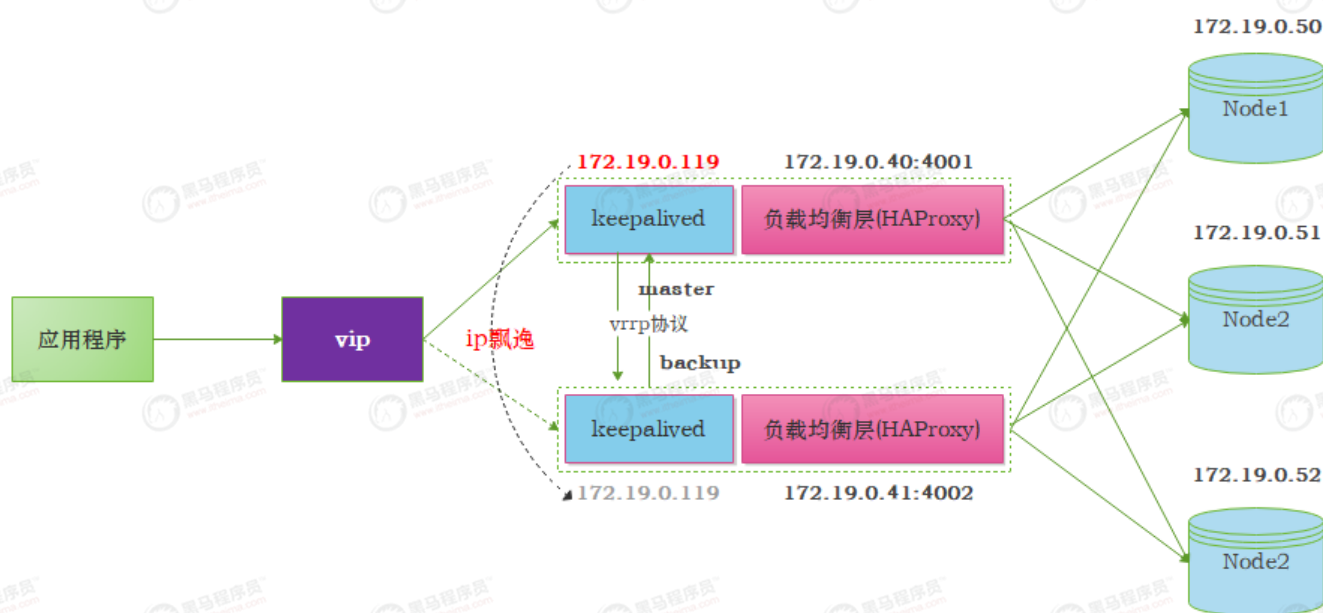
不具有负载均衡的连接情况：



具有负载均衡的连接情况：



本次我们所选择的负载均衡层的软件是HAProxy。为了保证负载均衡层的高可用，我们需要使用使用到 keepalived 软件，使用 vrrp 协议产生虚拟 ip 实现动态的 ip 飘逸。



keepalived 是以 VRRP 协议为实现基础的，VRRP 全称 Virtual Router Redundancy Protocol，即 [虚拟路由冗余协议](#)。虚拟路由冗余协议，可以认为是实现路由器高可用的协议，即将 N

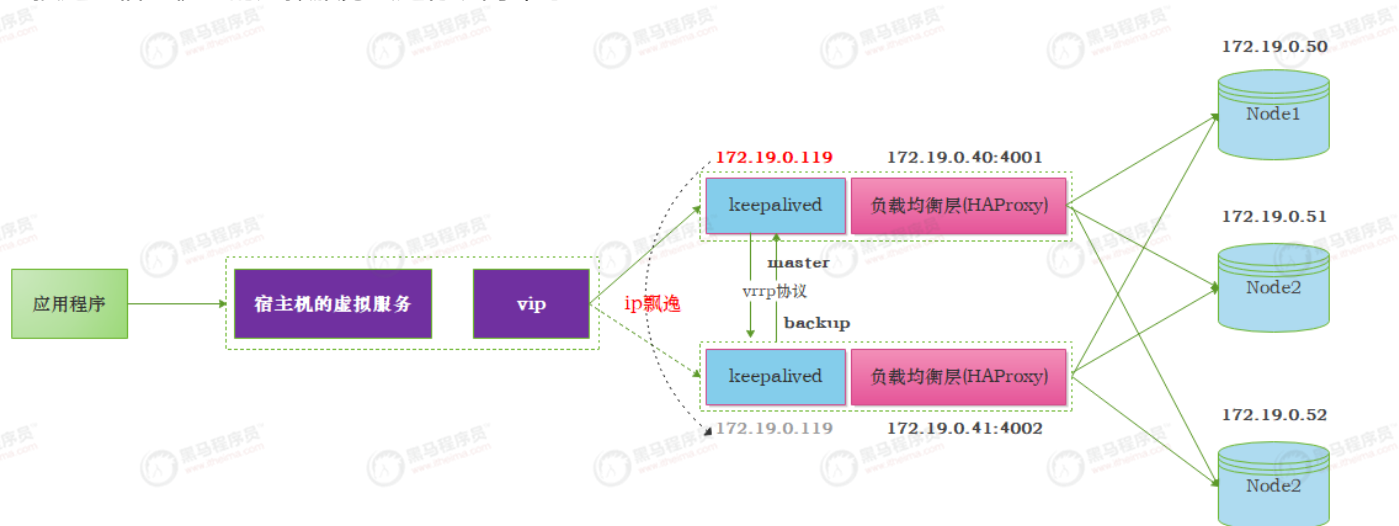
台提供相同功能的路由器组成一个路由器组，这个组里面有一个 master 和多个 backup，master 上面有一个对外提供服务的 vip（该路由器所在局域网内其他机器的默认路由为该 vip）

，master 会定义向 backup 发送 vrrp 协议数据包，当 backup 收不到 vrrp 包时就认为 master 宕掉了，这时就需要根据 VRRP 的优先级来选举一个 backup 当 master。这样的话就可以保证

路由器的高可用了。

由于我们的虚拟 ip 是在 docker 中创建的，而我们的应用程序是基于 windows 平台开发的，因此是无法直接使用 docker 容器中的虚拟 ip。因此我们需要在宿主机上安装 keepalived，在

宿主机上安装keepalived的主要目的是为了让keepalived产生虚拟服务，后期我们在访问HAProxy的时候，直接通过宿主机的虚拟服务去进行访问即可。



优化实现

HAProxy环境搭建

1、拉取镜像

```
1 | docker pull haproxy:1.7
```

2、创建一个HAProxy的配置文件haproxy.cfg

```
1 global
2     #工作目录
3     chroot /usr/local/etc/haproxy
4     #日志文件, 使用rsyslog服务中local5日志设备 (/var/log/local5) , 等级info
5     log 127.0.0.1 local5 info
6     #守护进程运行
7     daemon
8 defaults
9     log 127.0.0.1 local0 err #[err warning info debug]
10    mode http                #默认的模式mode { tcp|http|health }, tcp是4
                              #层, http是7层, health只会返回OK
11    retries 2                #两次连接失败就认为是服务器不可用
12    option redispatch        #当serverId对应的服务器挂掉后, 强制定向到其他健康
                              #的服务器
13    option abortonclose      #当服务器负载很高的时候, 自动结束掉当前队列处理比
                              #较久的链接
14    option dontlognull       #日志中不记录负载均衡的心跳检测记录
15    maxconn 4096             #默认的最大连接数
16    timeout connect 5000ms   #连接超时
17    timeout client 30000ms   #客户端超时
18    timeout server 30000ms   #服务器超时
19    #timeout check 2000      #=心跳检测超时
20
21    ##### 监控界面配置 #####
22 listen admin_stats
23     #监控界面的访问的IP和端口
24     bind 0.0.0.0:8888
25     #访问协议
26     mode http
27     #URI相对地址
28     stats uri /dbs
29     #统计报告格式
30     stats realm Global\ statistics
31     #登陆帐户信息
32     stats auth admin:admin
33
34 # rabbitmq管理界面配置
```

```

35 listen proxy_rabbitmq_web
36     #访问的IP和端口
37     bind 0.0.0.0:5000
38     #网络协议
39     mode tcp
40     #负载均衡算法（轮询算法）
41     #轮询算法: roundrobin
42     #权重算法: static-rr
43     #最少连接算法: leastconn
44     #请求源IP算法: source
45     balance roundrobin
46     # 这里是容器中的IP地址，由于配置的是轮询roundrobin, weight 权重其实没有生效
47     server rabbitmq_01 172.19.0.50:15672 check weight 1 maxconn 2000
48     server rabbitmq_02 172.19.0.51:15672 check weight 1 maxconn 2000
49     server rabbitmq_03 172.19.0.52:15672 check weight 1 maxconn 2000
50     # 使用keepalive检测死链
51     option tcpka
52
53 # rabbitmq服务代理，负载均衡配置
54 listen proxy_rabbitmq
55     #访问的IP和端口
56     bind 0.0.0.0:5010
57     #网络协议
58     mode tcp
59     #负载均衡算法（轮询算法）
60     #轮询算法: roundrobin
61     #权重算法: static-rr
62     #最少连接算法: leastconn
63     #请求源IP算法: source
64     balance roundrobin
65     # 这里是容器中的IP地址，由于配置的是轮询roundrobin, weight 权重其实没有生效
66     server rabbitmq_01 172.19.0.50:5672 check weight 1 maxconn 2000
67     server rabbitmq_02 172.19.0.51:5672 check weight 1 maxconn 2000
68     server rabbitmq_03 172.19.0.52:5672 check weight 1 maxconn 2000
69     # 使用keepalive检测死链
70     option tcpka

```

3、创建haproxy容器

```

1 docker run -di --network=docker-network --ip=172.19.0.40 -p 4001:8888 -p
5001:5000 -p 5010:5010 -v /usr/local/haproxy/haproxy-
01/config:/usr/local/etc/haproxy --name=haproxy_01 --privileged=true
haproxy:1.7 /bin/bash
2 docker run -di --network=docker-network --ip=172.19.0.41 -p 4002:8888 -p
5002:5000 -p 5011:5010 -v /usr/local/haproxy/haproxy-
02/config:/usr/local/etc/haproxy --name=haproxy_02 --privileged=true
haproxy:1.7 /bin/bash

```

4、进入到容器中，启动HAProxy

```

1 docker exec -it haproxy_01 /bin/bash # 进入容器
2 haproxy -f /usr/local/etc/haproxy/haproxy.cfg # 启动容器

```

5、通过浏览器就可以访问haproxy的后端管理界面了：<http://192.168.23.131:4002/dbs>,然后输入配置的用户名和密码就可以看到如下界面

admin_stats																													
	Queue			Session rate			Sessions			Total	LbTot	Last	Bytes		Denied	Req	Errors Conn	Resp	Warnings Retr	Redis	Status	LastChk	Wght	Server					
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit				In	Out										Act	Bck	Chk	Dwn	Downtime	Thrtle
Frontend	0	0	0	0	2	-	1	2	4 096	2	0	0	466	261	0	0	0	0	0	0	OPEN								
Backend	0	0	0	0	0	0	0	0	410	0	0	0s	466	261	0	0	0	0	0	0	42s UP			0	0	0		0	
proxy_rabbitmq_web																													
	Queue			Session rate			Sessions			Total	LbTot	Last	Bytes		Denied	Req	Errors Conn	Resp	Warnings Retr	Redis	Status	LastChk	Wght	Server					
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit				In	Out										Act	Bck	Chk	Dwn	Downtime	Thrtle
Frontend	0	0	0	0	0	-	0	0	4 096	0	0	0	0	0	0	0	0	0	0	0	OPEN								
rabbitmq_01	0	0	-	0	0	0	0	0	2000	0	0	?	0	0	0	0	0	0	0	0	42s UP	L4OK in 0ms	1	Y	-	0	0	0s	-
rabbitmq_02	0	0	-	0	0	0	0	0	2000	0	0	?	0	0	0	0	0	0	0	0	42s UP	L4OK in 0ms	1	Y	-	0	0	0s	-
rabbitmq_03	0	0	-	0	0	0	0	0	2000	0	0	?	0	0	0	0	0	0	0	0	42s UP	L4OK in 0ms	1	Y	-	0	0	0s	-
Backend	0	0	0	0	0	0	0	0	410	0	0	?	?	0	0	0	0	0	0	0	42s UP		3	3	0		0	0s	
proxy_rabbitmq																													
	Queue			Session rate			Sessions			Total	LbTot	Last	Bytes		Denied	Req	Errors Conn	Resp	Warnings Retr	Redis	Status	LastChk	Wght	Server					
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit				In	Out										Act	Bck	Chk	Dwn	Downtime	Thrtle
Frontend	0	0	0	0	0	-	0	0	4 096	0	0	0	0	0	0	0	0	0	0	0	OPEN								
rabbitmq_01	0	0	-	0	0	0	0	0	2000	0	0	?	0	0	0	0	0	0	0	0	42s UP	L4OK in 2ms	1	Y	-	0	0	0s	-
rabbitmq_02	0	0	-	0	0	0	0	0	2000	0	0	?	0	0	0	0	0	0	0	0	42s UP	L4OK in 0ms	1	Y	-	0	0	0s	-
rabbitmq_03	0	0	-	0	0	0	0	0	2000	0	0	?	0	0	0	0	0	0	0	0	42s UP	L4OK in 0ms	1	Y	-	0	0	0s	-
Backend	0	0	0	0	0	0	0	0	410	0	0	?	?	0	0	0	0	0	0	0	42s UP		3	3	0		0	0s	

HAProxy容器中安装keepalived

1、进入haproxy_01服务

```
1 | docker exec -it haproxy_01 /bin/bash
```

2、安装keepalived软件

```
1 | apt-get update # 更新软件列表,apt-get源不太稳定,建议多执行几次
2 | apt-get install keepalived # 安装keepalived软件
```

3、安装其他软件,供后期使用

```
1 | apt-get install net-tools # 安装ifconfig命令
```

4、创建一个keepalived.conf文件

vim /etc/keepalived/keepalived.conf, 内容如下所示:

```
1 | ! Configuration File for keepalived
2 |
3 | vrrp_instance VI_1 {
4 |
5 |     state MASTER # 标示状态为MASTER 备份机为BACKUP
6 |     interface eth0 # 定义虚拟网卡
7 |     virtual_router_id 100 # 定义组vriid, 同一组中
   | virtual_router_id必须相同
8 |     priority 100 # MASTER权重要高于BACKUP 比如
   | BACKUP为99
```

```

9      advert_int 1                                # MASTER 与 BACKUP 负载均衡器之间
      同步检查的时间间隔，单位是秒
10     authentication {                            # 定义组用户密码
11         auth_type PASS
12         auth_pass 123456
13     }
14
15     virtual_ipaddress {                          #定义docker内ip地址，必须要在和
      haproxy同一个网段
16         172.19.0.119
17     }
18
19 }

```

5、启动keepalived服务

```
1 | service keepalived start
```

6、查看eth0上是否已经绑定了虚拟ip

```
1 | ip add show eth0
```

haproxy_02容器中的keepalived软件的配置文件，如下所示：

```

1  ! Configuration File for keepalived
2
3  vrrp_instance VI_1 {
4
5      state BACKUP                                # 标示状态为MASTER 备份机为BACKUP
6      interface eth0                             # 定义虚拟网卡
7      virtual_router_id 100                      # 定义组vriid， 同一组中
      virtual_router_id必须相同
8      priority 99                                # MASTER权重要高于BACKUP 比如
      BACKUP为99
9      advert_int 1                                # MASTER 与 BACKUP 负载均衡器之间
      同步检查的时间间隔，单位是秒
10     authentication {                            # 定义组用户密码
11         auth_type PASS
12         auth_pass 123456
13     }
14
15     virtual_ipaddress {                          #定义docker内ip地址，必须要在和
      haproxy同一个网段

```

```
16      172.19.0.119
17    }
18
19 }
```

宿主机中keepalived

1、安装keepalived

```
1 | yum install -y keepalived
```

2、更改keepalived的配置文件的内容

```
1 | > /etc/keepalived/keepalived.conf # 清空原有配置文件的内容
2 | vim /etc/keepalived/keepalived.conf # 编辑配置文件，配置文件中的内容
   如下所示
```

```
1 | ! Configuration File for keepalived
2 | vrrp_instance VI_1 {
3 |     state MASTER
```

```

4 interface ens33 # 这里是宿主机的网卡，可以通过ip a查看
5     virtual_router_id 50
6     priority 100
7     advert_int 1
8     authentication {
9         auth_type PASS
10        auth_pass 1111
11    }
12    virtual_ipaddress { # 可以不用指定虚拟ip
13
14    }
15 }
16
17 # 虚拟服务器地址 IP 对外提供服务的端口
18 virtual_server 192.168.23.131 4000 {
19     delay_loop 3 # 健康检查时长 单位秒
20     lb_algo rr # 负载均衡算法
21     rr|wrr|lc|wlc|lb|c|sh|dh: LVS调度算法
22     lb_kind NAT # 负载均衡转发规则
23     persistence_timeout 50 # http服务会话时长 单位秒
24     protocol TCP # 健康检查用的是TCP还是UDP
25
26     # 对应后端真实的docker服务的ip地址和端口号
27     real_server 172.19.0.119 8888 { # 对应
28
29     # HAProxy的虚拟ip地址和后端管理系统端口
30     weight 1
31     }
32 }
33
34 # rabbitmq的web管理端的虚拟服务器
35 virtual_server 192.168.23.131 15672 {
36     delay_loop 3 # 健康检查时长 单位秒
37     lb_algo rr # 负载均衡算法
38     rr|wrr|lc|wlc|lb|c|sh|dh: LVS调度算法
39     lb_kind NAT # 负载均衡转发规则
40     persistence_timeout 50 # http服务会话时长 单位秒
41     protocol TCP # 健康检查用的是TCP还是UDP
42 }

```

```

40      # 对应后端真实的docker服务的ip地址和端口号
41      real_server 172.19.0.119 5000 {                                # 对应HAProxy的虚
拟ip地址和后端管理系统端口
42          weight 1
43      }
44  }
45  }
46
47  # rabbitmq的虚拟服务器
48  virtual_server 192.168.23.131 5672 {
49      delay_loop 3
50      lb_algo rr
51      lb_kind NAT
52      persistence_timeout 50
53      protocol TCP
54
55      # 对应后端真实的docker服务的ip地址和端口号
56      real_server 172.19.0.119 5010 {                                # 对应HAProxy的虚
拟ip地址和后端rabbitmq的监听端口
57          weight 1
58      }
59  }
60  }

```

3、启动keepalived服务，关闭防火墙

```

1  systemctl start keepalived
2  systemctl status firewalld.service

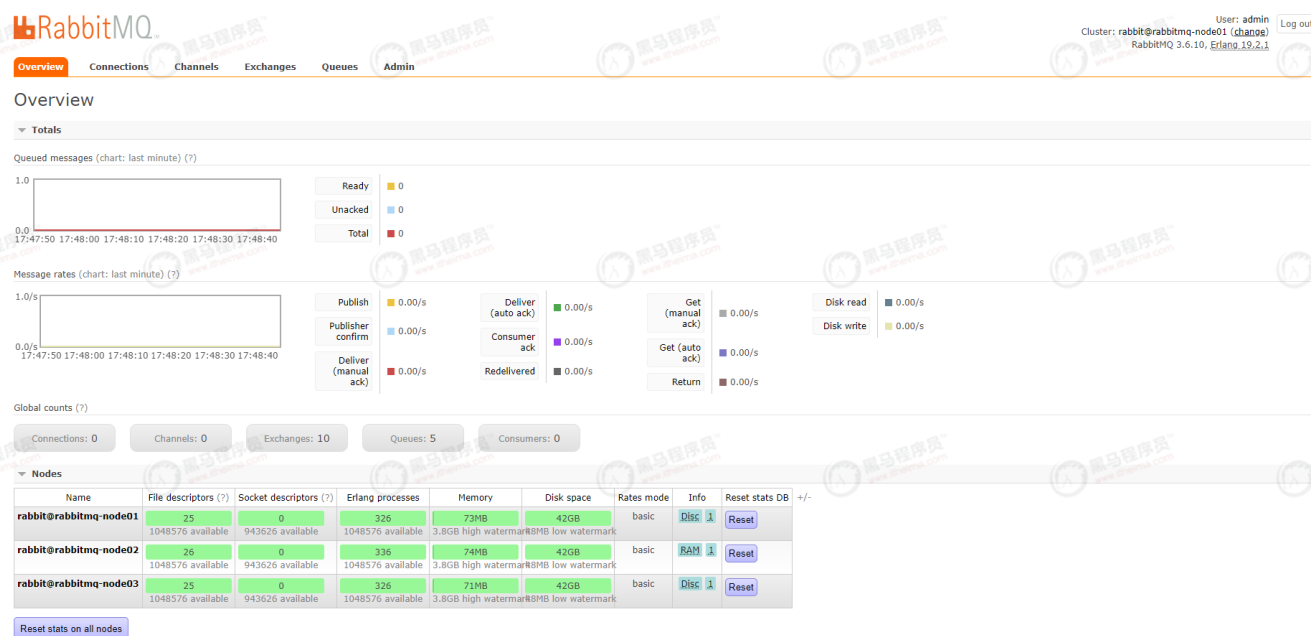
```

集群监控

RabbitMQ自带的Web管理端的插件

RabbitMQ作为一款非常成熟的消息中间件,必然少不了监控功能, RabbitMQ提供了Web版的页面监控（只在本地的浏览器端访问地址：<http://xxx.xxx.xxx.xxx:15672/>，默认端口号是15672）

Web监控页面如下图所示：



如果只是在测试或生产环境小规模地应用RabbitMQ消息队列（比如业务并发访问量较低），那么简单地用RabbitMQ自带的Web页面进行监控也就足够了。但是，如果对RabbitMQ的并发性

能、高可用和其他参数都有一些定制化的监控需求的话，那么我们就有必要采用其他方式来达到该目标。

RabbitMQ的tracing消息轨迹追踪

在使用Rabbitmq的时候，如果我们想清楚的知道消息的投递过程，那么此时我们就可以使用这个tracing插件来完成。使用步骤如下所示：

1、开启rabbitmq的tracing插件

```
1 rabbitmq-plugins enable rabbitmq_tracing
```

开启tracing插件mq的web页面可以看到admin标签页多了一个选项



二、配置追踪

如下图指定vhost添加一个追踪



Format的取值：Text, json

Pattern规则说明：

#匹配所有的消息，无论是发布还是消费的信息

publish.# 匹配所有发布的消息。

deliver.# 匹配所有被消费的消息。

#.test 如果test是队列，则匹配已经被消费了的test队列的消息。如果test是exchange，则匹配所有经过该exchange的消息。

注意：如果是json格式的日志，消息体会采用Base64进行编码。

采用RabbitMQ的HTTP API接口进行监控

要构建独立的监控系统，可以使用RabbitMQ本身提供的Restful HTTP API接口来获取各种业务监控需要的实时数据。当然，这个接口的作用远不止于获取一些监控数据，也可以通过这些HTTP API来操作RabbitMQ进行各种集群元数据的添加/删除/更新的操作。

下面列举了可以利用RabbitMQ的HTTP API接口实现的各种操作：

HTTP API URL	HTTP 请求类型	接口含义
/api/connections	GET	获取当前RabbitMQ集群下所有打开的连接
/api/nodes	GET	获取当前RabbitMQ集群下所有节点实例的状态信息
/api/vhosts/{vhost}/connections	GET	获取某一个虚拟机主机下的所有打开的connection连接
/api/connections/{name}/channels	GET	获取某一个连接下所有的管道信息
/api/vhosts/{vhost}/channels	GET	获取某一个虚拟机主机下的管道信息
/api/consumers/{vhost}	GET	获取某一个虚拟机主机下的所有消费者信息
/api/exchanges/{vhost}	GET	获取某一个虚拟机主机下面的所有交换器信息
/api/queues/{vhost}	GET	获取某一个虚拟机主机下的所有队列信息
/api/users	GET	获取集群中所有的用户信息
/api/users/{name}	GET/PUT/DELETE	获取/更新/删除指定用户信息
/api/users/{user}/permissions	GET	获取当前指定用户的所有权限信息
/api/permissions/{vhost}/{user}	GET/PUT/DELETE	获取/更新/删除指定虚拟主机下特定用户的权限
/api/exchanges/{vhost}/{name}/publish	POST	在指定的虚拟机主机和交换器上发布一个消息
/api/queues/{vhost}/{name}/get	POST	在指定虚拟机主机和队列名中获取消息，同时该动作会修改队列状态
/api/healthchecks/node/{node}	GET	获取指定节点的健康检查状态

上面的HTTP API接口只是列举了RabbitMQ所支持的部分功能，读者可以参考RabbitMQ官方文档和访问<http://192.168.23.131:15672/api/>的Web页面来获取更多的其他接口信息。业务研发的

同学可以使用Apache的httpcomponents组件—HttpClient或者Spring的RestTemplate组件生成并发送HTTP的GET/POST/DELETE/PUT请求至RabbitMQ Server，根据自己的业务目标完成相应的业务监控需求。

实例代码：RabbitmqMonitorApplication类

注意：默认的交换机使用 %2F

2.6.2 RabbitMQ镜像集群

镜像队列原理

思考的问题：

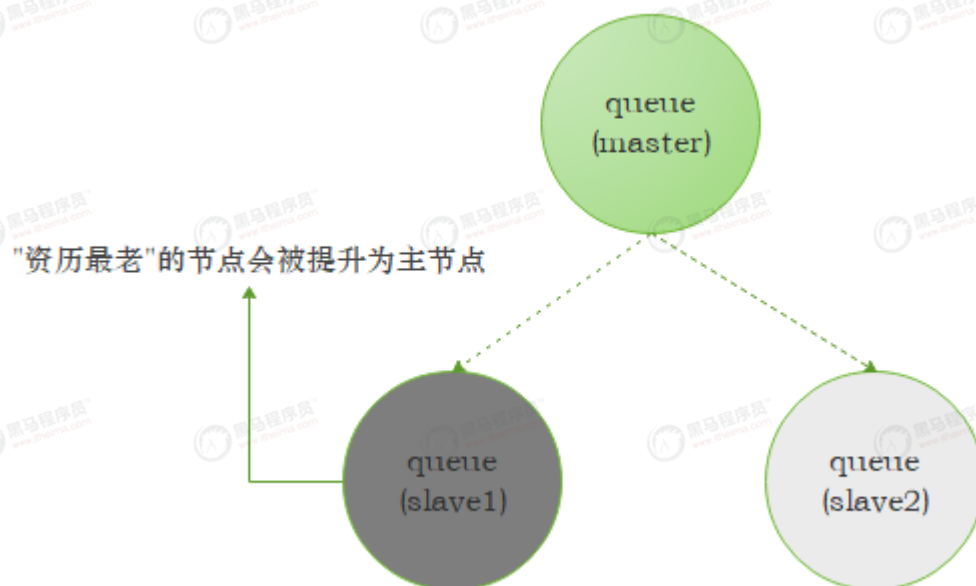
1、为什么要存在镜像队列

为了保证队列和消息的高可用

2、什么是镜像队列，镜像队列是如何进行选取主节点的？

引入镜像队列的机制，可以将队列镜像到集群中的其他的Broker节点之上，如果集群中的一个节点失效了，队列能自动的切换到镜像中的另一个节点之上以保证服务的可用性。在通常

的用法中，针对每一个配置镜像的队列（一下称之为镜像队列）都包含一个主节点(master)和若干个从节点(slave)，如下图所示：

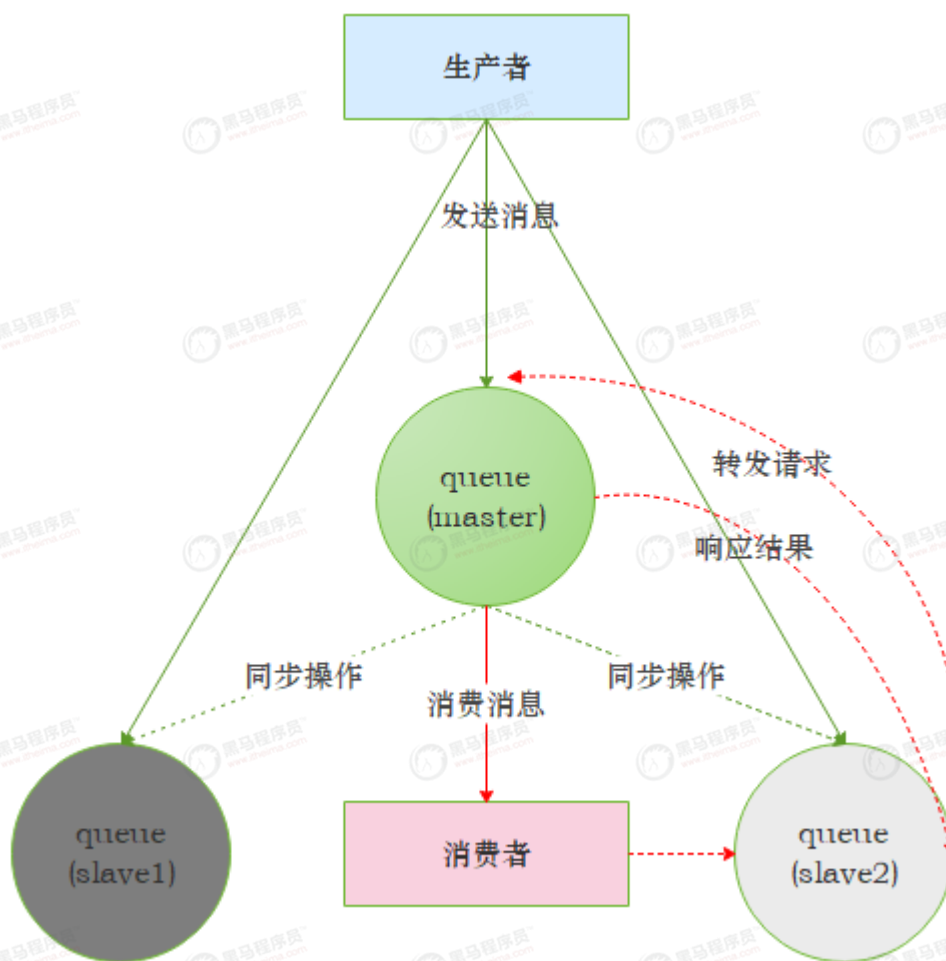


slave会准确地按照master执行命令顺序进行动作，故slave和master上维护的状态应该也是相同的。如果master由于某种原因宕机了，那么“资源最老”的slave会被提升为新的master。

根据slave加入的时间排序，时间最长的slave即为“资历最老”。发送到镜像队列的所有消息会被同时发往master和所有的slave，如果此时master挂掉了，消息还会在slave上，这样slave提升为master的时候消息也不会丢失。

3、镜像队列的工作模式是什么？

如下图所示：



除发送消息（Basic.Publish）外的所有动作都只会向master发送，然后在由master将命令执行的结果广播给各个slave。如果消费者与slave建立连接并进行订阅消息，其实质上都是从master

上获取消息，只不过看似是从slave上消费而已。比如：消费者与slave建立了TCP连接之后执行一个Basic.GET的操作，那么首先是有slave将Basic.GET请求发往master，再由master准备好数

据返回给slave，最后又slave投递给消费者。

镜像策略介绍

针对某一个队列去配置其对应的镜像其实比较简单，我们只需要去添加一个镜像策略即可：

```
1 rabbitmqctl set_policy [-p <vhost>] [--priority <priority>] [--apply-to <apply-to>] <name> <pattern> <definition>
```

指令参数详情:

参数名称	描述
-p	可选参数，针对指定 vhost 下的exchange或 queue
-priority	可选参数，policy 的优先级
-apply-to	可选参数，策略适用的对象类型，其值可为“queues”，“exchanges”或“all”默认是“all”
name	policy 的名称
pattern	匹配模式（正则表达式）
definition	镜像定义，json 格式，包括三部分（ha-mode,ha-params,ha-sync-mode）具体配置见下表

definition参数详情:

参数名称	描述
ha-mode	指定镜像队列模式，其值可为"all","exactly"或"nodes"，all：表示在集群所有节点上进行镜像；exactly：表示在指定个数的节点上镜像，节点个数由ha-params指定；nodes：表示在指定节点上进行镜像，节点名称通过ha-params指定。
ha-params	ha-mode模式需要用到的参数：exactly 模式下为数字表述镜像节点数，nodes 模式下为节点列表表示需要镜像的节点。
ha-sync-mode	镜像队列中消息的同步方式，其值可为"automatic"或"manually"。

实例代码：

例如：对队列名称为 hello 开头的所有队列镜像镜像，并且在集群的节点 rabbit@10.18.195.57上进行镜像，队列消息自动同步，policy 的设置命令：

```
1 rabbitmqctl set_policy --apply-to queues hello-ha "^hello" '{"ha-mode":"nodes","ha-params":["rabbit@10.18.195.57"],"ha-sync-mode":"automatic"}'
```

我们也可以通过rabbitmq的后台管理系统来添加镜像策略，如下图所示：

The screenshot shows the RabbitMQ Admin interface. The 'Policies' tab is selected. The 'Add / update a policy' form is visible, with a red box highlighting the input fields. The form includes:

- Name: policy_test01
- Pattern: ^queue_test01
- Apply to: Queues
- Priority: (empty)
- Definition:
 - ha-mode: all
 - ha-params: String
 - ha-sync-mode: automatic

Below the form, there are links for 'Add policy' and a list of existing policies.

添加完毕以后，我们再次查看队列的定义信息，如下所示：

Overview

Connections

Channels

Exchanges

Queues

Admin

Queues

▼ All queues (2)

Pagination

Page

1

 of 1 - Filter:

☐ Regexp (?) (?)

Displaying 2 items, page size up to: 100

Overview				Messages			Message rates			
Name	Node	Features	State	Ready	Unacked	Total	incoming	deliver	get	ack
queue_test01	rabbitmq-node01	2 D policy_test01	idle	0	0	0	0.00/s	0.00/s	0.00/s	0.00/s
queue_test02	rabbitmq-node02	D	idle	0	0	0	0.00/s	0.00/s	0.00/s	0.00/s

➤ Add a new queue

此时也就证明我们的镜像队列配置成功了。

镜像队列测试

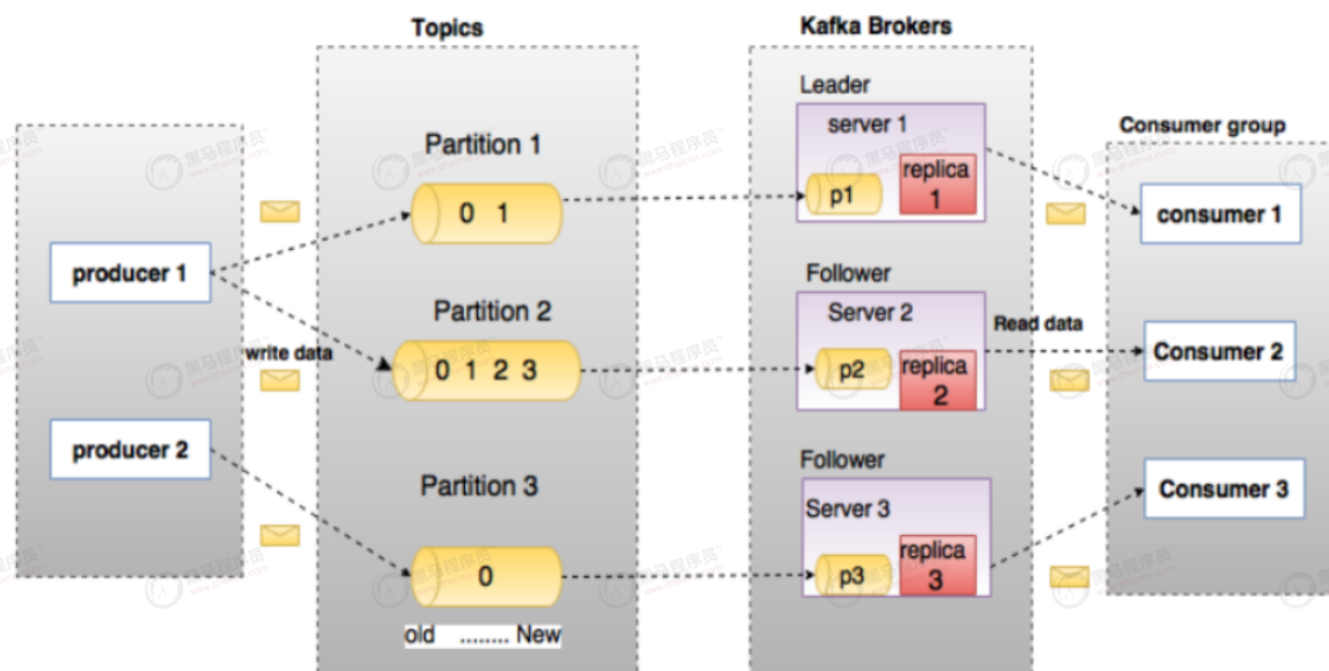
关闭某一个队列进程所在的节点，然后使用该队列进行消息的收发操作。

3 第三章: Kafka在秒杀场景下进阶

3.1 Kafka基础知识回顾

3.1.1 kafka架构回顾

kafka的具体架构如下所示:



Producer

生产者即数据的发布者, 该角色将消息发布到Kafka的topic中。broker接收到生产者发送的消息后, broker将该消息追加到当前用于追加数据的segment文件中。

生产者发送的消息, 存储到一个partition中, 生产者也可以指定数据存储的partition。

Consumer

消费者可以从broker中读取数据。消费者可以消费多个topic中的数据。

Topic

在Kafka中, 使用一个类别属性来划分数据的所属类, 划分数据的这个类称为topic。如果把Kafka看做为一个数据库, topic可以理解为数据库中的一张表, topic的名字即为表名。

Partition

topic中的数据分割为一个或多个partition。每个topic至少有一个partition。每个partition中的数据使用多个segment文件存储。partition中的数据是有序的, partition间的数据丢失了数据

的顺序。如果topic有多个partition, 消费数据时就不能保证数据的顺序。在需要严格保证消息的消费顺序的场景下, 需要将partition数目设为1。

Partition offset

每条消息都有一个当前Partition下唯一的64字节的offset，它指明了这条消息的位置。

Replicas of partition

副本是一个分区的备份。副本不会被消费者消费，副本只用于防止数据丢失，即消费者不从为follower的partition中消费数据，而是从为leader的partition中读取数据。副本之间是一主多从的关系。

Broker

Kafka 集群包含一个或多个服务器，服务器节点称为broker。broker存储topic的数据。如果某topic有N个partition，集群有N个broker，那么每个broker存储该topic的一个partition。如果

某topic有N个partition，集群有(N+M)个broker，那么其中有N个broker存储该topic的一个partition，剩下的M个broker不存储该topic的partition数据。如果某topic有N个partition，集群

中broker数目少于N个，那么一个broker存储该topic的一个或多个partition。在实际生产环境中，尽量避免这种情况的发生，这种情况容易导致Kafka集群数据不均衡。

Leader

每个partition有多个副本，其中有且仅有一个作为Leader，Leader是当前负责数据的读写的partition。

Follower

Follower跟随Leader，所有写请求都通过Leader路由，数据变更会广播给所有Follower，Follower与Leader保持数据同步。如果Leader失效，则从Follower中选举出一个新的Leader。

当Follower与Leader挂掉、卡住或者同步太慢，leader会把这个follower从“in sync replicas”（ISR）列表中删除，重新创建一个Follower。

Zookeeper

Zookeeper负责维护和协调broker。当Kafka系统中新增了broker或者某个broker发生故障失效时，由ZooKeeper通知生产者和消费者。生产者和消费者依据Zookeeper的broker状态信息与broker协调数据的发布和订阅任务。

AR(Assigned Replicas)

分区中所有的副本统称为AR。

ISR(In-Sync Replicas)

所有与Leader部分保持一致的副（包括Leader副本在内）本组成ISR。

OSR(Out-of-Sync-Replicas)

与Leader副本同步滞后过多的副本。

3.1.2 环境搭建以及测试

环境搭建参考文档《[Docker部署Kafka](#)》，导入基础工程进行测试。

3.2 Kafka消息可靠性传输

3.2.1 第一种情况

消费端消息丢失

场景描述

位移提交：对于Kafka中的分区而言，它的每条消息都有唯一的offset，用来表示消息在分区中的位置。对于消费者而言，它也有一个offset的概念，消费者使用offset来表示消费到分

区中某个消息所在的位置。单词"offset"可以编译为"偏移量"，也可以翻译为"位移"，在很多的中文资料中都会交叉使用"偏移量"和"位移"这两个词，对于消息在分区中的位置，我们将

offset称之为"偏移量"；对于消费者消费到的位置，将offset称为"位移"，有时候也会更明确地称之为"消费位移"（"偏移量"是在讲分区存储层面的内容，"位移"是在讲消费层面的内容）

当然对于一条消息而言，它的偏移量和消费者消费它时的消费位移是相等的。

当每一次调用poll()方法时，它返回的是还没有消费过的消息集（当然这个前提是消息以及存储在Kafka中了，并且暂不考虑异常情况的发生），要做到这一点，就需要记录上一次消费

时的消费位移。并且这个消费位移必须做持久化保存，而不是单单保存在内存中，否则消费者重启之后就无法知晓之前的消费位移。消费位移存储在Kafka内部的主题__consumer_offsets

中。这里把将消费位移存储起来的动作称之为"提交"，消费者在消费完消息之后需要执行消费位移的提交。

默认情况下Kafka的消费位移提交是**自动提交**，这个由消费者客户端参数enable.auto.commit配置，默认值为true。当然这个默认的自动提交并不是每消费一条消息就提交一次，而是

定期提交，这个定期的周期时间由客户端参数auto.commit.interval.ms配置，默认值为5秒。自动位移提交的动作是在poll()方法的逻辑里面完成的，在每次真正向服务端发起拉取请求

之前会检查是否可以位移提交，如果可以，那么就会提交上一次轮询的位移。

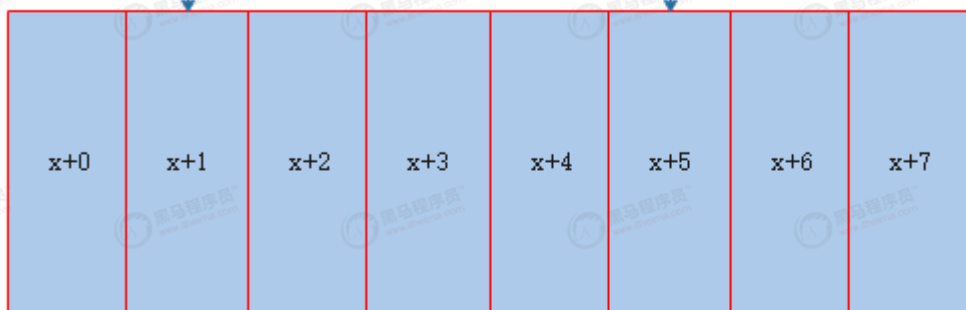
自动位移提交带来为问题：

- 1、重复消费
- 2、消息丢失

重复消费原理如下图所示：

第y次消息拉取

第y+1次消息拉取

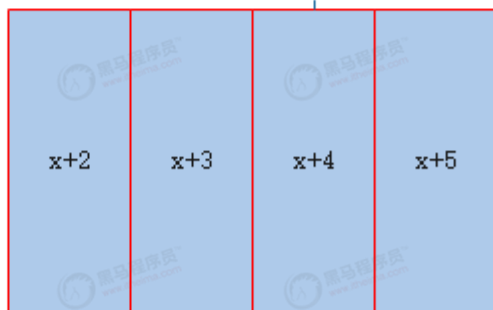


第y次位移提交



提交位移时间到了，在提交之前消费者宕机。
位移提交失败，在下次拉取消息的时候还是从上次提交位移的位置进行拉取

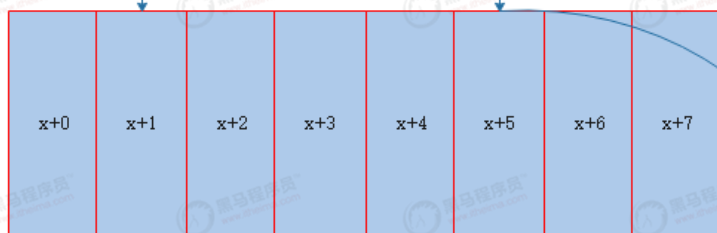
消费者线程消费消息



消息丢失如下图所示：

第y次消息拉取

第y+1次消息拉取

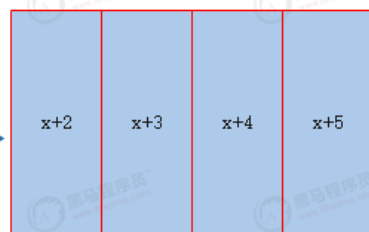


第y次位移提交

位移提交时间到了进行第y+1次位移提交

拉取消息线程

消费者线程进行消息消费，假设在进行第y+1次位移提交的时候，消费者线程执行到了x+4这个消息，此时发生了异常。等到消费者线程恢复以后，就可以从x+6这个消息开始进行拉取消费。这样x+5这个消息就丢失了



解决方案

解决方案：将自动位移提交更改为手动位移提交

注意：

- 1、在单独使用Kafka的java客户端将位移提交的模式更改为手动位移提交，那么我们就需要显示的调用consumer的方法完成位移提交（参考代码Kafka手动位移提交.java）。
- 2、在使用的是spring boot和Kafka进行整合，当我们将spring.kafka.consumer.enable-auto-commit的值设置为false以后，只有在特定的**提交模式**下我们可以手动进行提交。

在一些提交模式下由spring根据约定的条件控制提交。

常见的提交模式是在ContainerProperties.AckMode这个枚举类中定义。AckMode针对ENABLE_AUTO_COMMIT_CONFIG=false时生效，有以下几种：

- RECORD：每处理一条commit一次
- BATCH：每次poll的时候批量提交一次，频率取决于每次poll的调用频率
- TIME：每次间隔ackTime的时间去commit
- COUNT：累积达到ackCount次的ack去commit
- COUNT_TIME：ackTime或ackCount哪个条件先满足，就commit
- MANUAL：listener负责ack，但是背后也是批量上去
- MANUAL_IMMEDIATE：listener负责ack，每调用一次，就立即commit

spring配置如下:

```
1 # kafka配置
2 spring:
3     kafka:
4         consumer:
5             bootstrap-servers: 192.168.23.131:9092
6             enable-auto-commit: false # 设置手动位移提交
7             listener:
8                 ack-mode: manual_immediate
```

代码实现:

```
1 @KafkaListener(topics = "itheima" , groupId = "itheima.demo")
2 public void consumerHandler(String msg , KafkaConsumer consumer) {
3     LOGGER.info("consumer topic is : {} , msg is ----> {} " , "itheima"
4     , msg);
5     // consumer.commitSync(); // 同步位移提交
6     consumer.commitAsync((offsets , e) -> {
7         if(e == null){
8             LOGGER.info("位移提交成功....");
9         }else {
10             Set<Map.Entry<TopicPartition, OffsetAndMetadata>> entries =
11             offsets.entrySet();
12             for(Map.Entry<TopicPartition , OffsetAndMetadata> entry :
13             entries) {
14                 TopicPartition topicPartition = entry.getKey();
15                 OffsetAndMetadata offsetAndMetadata = entry.getValue();
16                 LOGGER.info("位移提交失败....topicPartition is {} ,
17                 offsets is {} " , topicPartition.partition() ,
18                 offsetAndMetadata.offset());
19             }
20         }
21     });
22 }
```

3.2.2 第二种情况

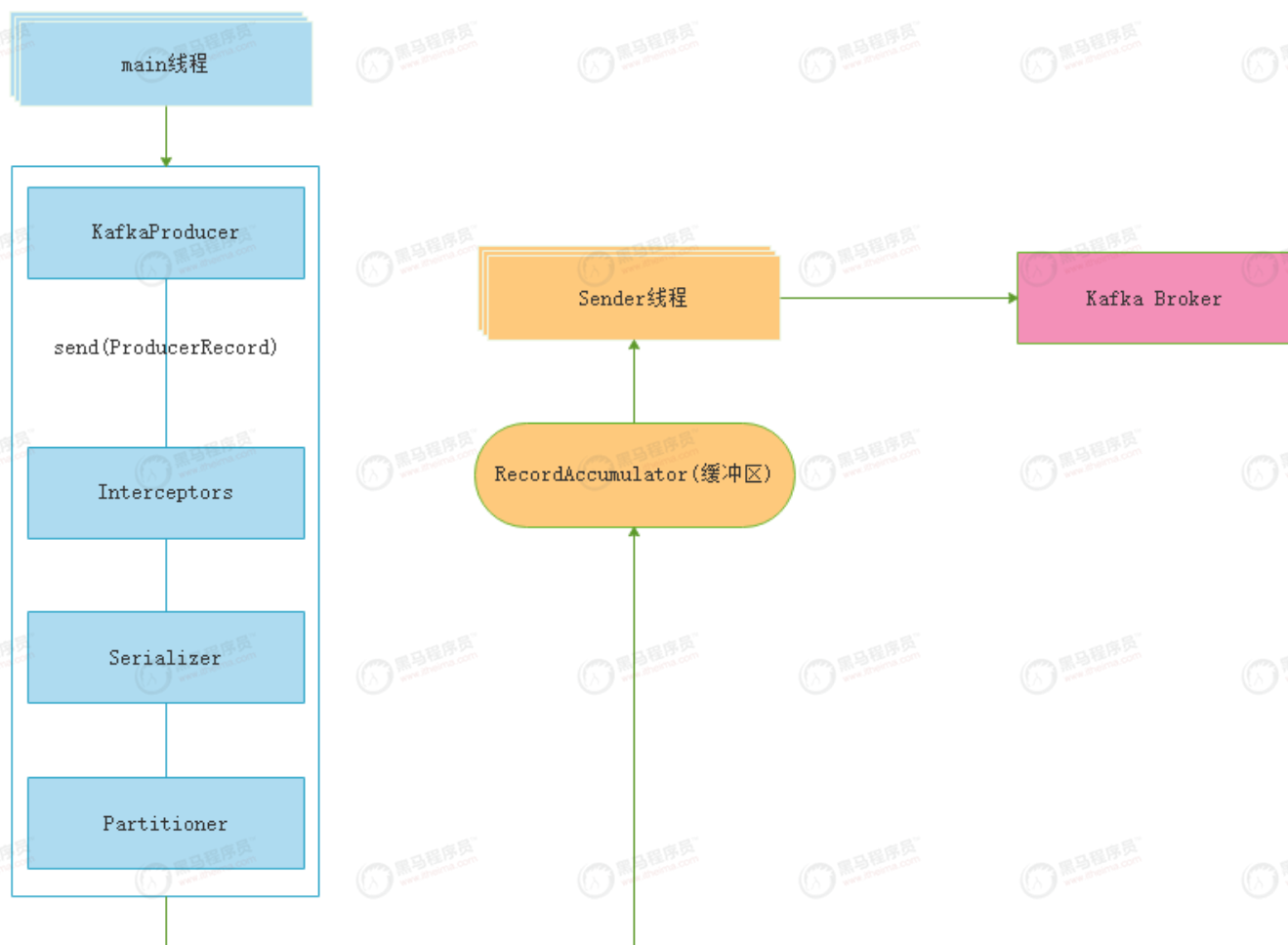
生产者将消息发送到Broker中以后，消息还没有被及时消费，此时Broker宕机了，这样就会导致消息丢失。

场景描述

Kafka消息的发送流程：

在消息发送的过程中，涉及到了两个线程——main线程和Sender线程，以及一个线程共享变量——RecordAccumulator。main线程将消息发送给RecordAccumulator，Sender线程

会根据指定的条件，不断从RecordAccumulator中拉取消息发送到Kafka broker。如下图所示：



Sender线程拉取消息的条件：

- 1、缓冲区大小达到一定的阈值（默认是16384byte），可以通过spring.kafka.producer.batch-size进行设定
- 2、缓冲区等待的时间达到设置的阈值（默认是0），可以通过linger.ms属性进行设定

发送消息的三种方式：1、发后即忘 2、同步消息发送 3、异步消息发送

- 发后即忘

只管往kafka发送消息（消息只发送到缓冲区）而并不关心消息是否正确到达。正常情况没什么问题，不过有些时候（比如不可重试异常）会造成消息的丢失。

这种发送方式性能最高，可靠性最差。如下所示：

```

1 // 演示消息发送： 发送即忘
2 public static void sendMessageMethod01() {
3     for(int x = 0 ; x < 5 ; x++) {
4         String msg = "Kafka环境测试...." + x ;
5         kafkaTemplate.send("itheima" , msg) ; // 发后即忘
6
7         LOGGER.info("send msg success ---> {} " , msg);
8     }
9 }

```

- 同步消息发送

其实kafkaTemplate.send方法并不是返回void，而是ListenableFuture<SendResult<K, V>>，该类继承了jdk concurrent包的Future。如下：

```

1 @Override
2 public ListenableFuture<SendResult<K, V>> send(String topic, @Nullable V
3 data) {
4     ProducerRecord<K, V> producerRecord = new ProducerRecord<>(topic,
5 data);
6     return doSend(producerRecord);
7 }

```

要实现同步发送的，可以利用返回的ListenableFuture实现，如下：

```

1 // 演示消息发送： 同步发送
2 public static void sendMessageMethod02() throws ExecutionException,
3 InterruptedException {
4     for(int x = 0 ; x < 5 ; x++) {
5         String msg = "Kafka环境测试...." + x ;
6         kafkaTemplate.send("itheima" , msg).get() ; // 同步消息发送
7
8         LOGGER.info("send msg success ---> {} " , msg);
9     }
10 }

```

实际上send()方法本身就是异步的，send()方法返回的Future对象可以使调用方稍后获得发送的结果。在执行send()方法之后直接链式调用了get () 方法可以阻塞等待Kafka的响应，

直到消息发送成功，或者发生异常。如果发生异常，那么就需要捕获异常并交由外层逻辑处理。这种方式性能最差，可靠性较好。

- 异步发送

在send方法里指定一个Callback的回调函数，Kafka在返回响应时调用该函数来实现异步的发送确认。异步发送方式的示例如下：

```
1 // 演示消息发送： 异步发送
2 public static void sendMessageMethod03() throws ExecutionException,
   InterruptedException {
3
4     for(int x = 0 ; x < 5 ; x++) {
5
6         // kafka消息的异步发送
7         String msg = "kafka环境测试...." + x ;
8         kafkaTemplate.send("itheima" , msg).addCallback((obj) -> {
9             LOGGER.info("send msg to kafka broker success ---> {} " ,
10 ((SendResult)obj).getProducerRecord().value());
11         } , (t) -> {
12             t.printStackTrace();
13         });
14         LOGGER.info("send msg to local cache success ---> {} " , msg);
15     }
16 }
```

这种方式的特点：性能较好，可靠性也有保障

解决方案

针对上述情况所产生的消息丢失，可以解决方案有如下几种：

- 1、给topic设置replication.factor参数大于1,要求每个partition必须最少有两个副本
- 2、搭建Kafka集群，让各个分区副本均衡的分配到不同的broker上
- 3、在producer端设置acks=all,要求每条数据写入replicas后,才认为写入成功
 - ack=0，生产者在成功写入消息之前不会等待任何来自服务器的响应。如果出现问题生产者是感知不到的，消息就丢失了。不过因为生产者不需要等待服务器响应，所以它可以以网络能够支持的最大速度发送消息，从而达到很高的吞吐量。
 - ack=1，默认值为1，只要集群的首领节点收到消息，生产者就会收到一个来自服务器的成功响应。如果消息无法达到首领节点（比如首领节点崩溃，新的首领还没有被选举出来），生产者会收到一个错误响应，为了避免数据丢失，生产者会重发消息（Kafka生产者内部机制）。但是，这样还有可能会导致数据丢失，如果收到写成功通知，此时首领节点还没来的及同步数据到follower节点，首领节点崩溃，就会导致数据丢失。
 - ack=-1/all，只有当所有参与复制的节点都收到消息时，生产者会收到一个来自服务器的成功响应，这种模式是最安全的，它可以保证不止一个服务器收到消息。

当生产者发送消息完毕以后，没有收到Broker返回的ack，此时就会触发重试机制或者抛出异常。我们可以通过retries参数设置重试次数（spring boot和Kafka整合默认的重试次数为0），

发送客户端会进行重试直到broker返回ack；当消息重试了指定次数以后，还没有收到服务端的ack，此时就会抛出异常。如果我们还需要保证消息的顺序性，那么我们就需要将

max.in.flight.requests.per.connection设置为1，该参数指定了生产者在收到服务器响应之前可以发送多少个消息。它的值越高，就会占用越多的内存，不过也会提升吞吐量。把它设为1可以保证消息是按照发送的顺序写入服务器的，即使发生了重试。

如何保证顺序性：如果把retries设为非零整数，同时把

max.in.flight.requests.per.connection 设为比1大的数，那么，如果第一个批次消息写入失败，而第二个批次写

入成功，broker会重试写入第一个批次。如果此时第一个批次也写入成功，那么两个批次的顺序就反过来了。

一般来说，如果某些场景要求消息是有序的，那么消息是否写入成功也是很关键的，所以不建议把retries设为0。可以把**max.in.flight.requests.per.connection**设为1，

这样在生产者尝试发送第一批消息时，就不会有其他的消息发送给broker。不过这样会严重影响生产者的吞吐量，所以只有在对消息的顺序有严格要求的情况下才能这么做。

配置如下所示：

```
1 spring:
2   kafka:
3     producer:
4       bootstrap-servers: 192.168.23.131:9092
5       acks: all
6       retries: 2
7       properties: {'max.in.flight.requests.per.connection' : 1}
```

3.3 Kafka秒杀公平性保证

消息的可靠性传输可以保证秒杀业务的公平性。关于秒杀业务的公平性，我们还需要考虑一点：消息的顺序性（先进入队列的消息先进行处理）

Kafka保证消息顺序性的特点如下所示：

topic中的数据分割为一个或多个partition。每个topic至少有一个partition。在单个partition中的数据是有序的，如果消息分散在不同的 partition，Kafka 无法保证其顺序性。但只需要确

保**要求顺序性的若干消息发送到同一个 partiton**，即可满足其顺序性。并且在进行消息消费的时候，需要确保消费者是进行单线程消费。

```
bin/kafka-topics.sh --zookeeper 172.19.0.60:2181 --create --topic itcast --partitions 3 --replication-factor 1 # 创建itcast分区测试
```

要保证若干消息发送到同一个partiton中，那么我们就需要在发送消息的时候指定一个分区的id，那么这样的话消息就被发送到同一个分区中。

```
1 // 发送消息到指定的分区，保证分区的消息顺序性
2 public static void sendMessageToDestPartition() {
3
4     for(int x = 0 ; x < 5 ; x++) {
5
6         // kafka消息的异步发送
7         String msg = "Kakfa环境测试...." + x ;
8         kafkaTemplate.send("itcast" ,0 , "order" ,
9 msg).addCallback((obj) -> {
10             LOGGER.info("send msg to kafka broker success ----> {} " ,
11 ((SendResult)obj).getProducerRecord().value());
12             } , (t) -> {
13                 t.printStackTrace();
14             });
15         LOGGER.info("send msg to local cache success ----> {} " , msg);
16     }
17 }
```

消费者进行指定分区的消费：

```
1 @KafkaListener(topicPartitions =
  {@org.springframework.kafka.annotation.TopicPartition(topic = "itcast" ,
  partitions = "0")}) , groupId = "itcast.demo")
2 public void consumerOrderMessageHandler(String msg , KafkaConsumer
  consumer) {
3     LOGGER.info("consumer topic is : {} , msg is ----> {} " , "itcast" ,
  msg);
4     consumer.commitAsync();
5 }
```

3.4 Kafka秒杀不超卖保证

3.4.1 生产端消息重复

问题描述

生产者发送的消息没有收到正确的broke响应，导致producer重试。producer发出一条消息，broker落盘以后因为网络等种种原因发送端得到一个发送失败的响应或者网络中断，

然后producer收到一个可恢复的Exception重试消息导致消息重复。

解决方案

解决方案：

1、启动kafka的幂等性

2、retries=0，不重试（可能会丢消息，适用于吞吐量指标重要性高于数据丢失，例如：日志收集）

所谓幂等性，就是对接口的多次调用所产生的结果和调用一次是一致的。生产者在进行重试的时候有可能会重复写入消息，而使用Kafka的幂等性功能就可以避免这种情况。

开启幂等性的方式比较简单，我们只需要设置**enable.idempotence**参数为**true**就可以了。如下所示：

```
1 spring:
2   kafka:
3     producer:
4       bootstrap-servers: 192.168.23.131:9092
5       acks: all
6       retries: 2
7       properties: {'max.in.flight.requests.per.connection' : 1 ,
  "enable.idempotence" : true }
```

如果使用幂等性，并且我们显示的指定了**retries**，**acks**，**max.in.flight.requests.per.connection**这几个参数，那么就对这几个参数的配置是有要求的：

retries的值必须是大于0，如果设置不对就会抛出如下异常：

```
1 Caused by: org.apache.kafka.common.config.ConfigException: Must set
  retries to non-zero when using the idempotent producer.
```

max.in.flight.requests.per.connection的值不能大于5，如果设置不对就会抛出如下异常：

```
1 Caused by: org.apache.kafka.common.config.ConfigException: Must set
  max.in.flight.requests.per.connection to at most 5 to use the idempotent
  producer.
```

acks的取值需要设置为-1/all，如果设置不对就会抛出如下异常：

1 Caused by: org.apache.kafka.common.config.ConfigException: Must set acks to all in order to use the idempotent producer. Otherwise we cannot guarantee idempotence.

幂等性原理介绍:

为了实现生产者幂等性, Kafka为此引入了producer id(PID) 和序列号 (sequence number) 这两个概念, 每个新的生产者实例在初始化的时候都会被分配一个PID, 这个PID对用户而言

是完全透明的。对于每个PID, 消息发送到的每一个分区都有对应的序列号, 这些序列号从0开始单调递增。生产者每发送一条消息就会将<PID, 分区>对应的序号的值加1。

broker端会在内存中为每一对<PID, 分区>维护一个序列号。对于收到的每一条消息, 会存在这样的几种情况:

1、 $SN_{new} = SN_{old} + 1$ 时, broker才会接收它。

2、 $SN_{new} < SN_{old} + 1$, 那么说明消息被重复写入, broker可以直接将其丢弃。

3、 $SN_{new} > SN_{old} + 1$, 那么说明中间有数据尚未写入, 出现了乱序, 暗示可能有消息丢失, 对应的生产者会抛出OutOfOrderSequenceException, 这个异常是一个严重的异常。

幂等性不能跨分区实现。

注: Kafka的幂等性只能保证单个生产者会话(session)中单分区的幂等。

相关知识

幂等性并不能跨多个分区运作，比如我们现在要想发送3个消息，当第二个消息发送完毕以后程序报错了，这样第三个消息就没有发送成功，当下一次在调用这个方法发送数据的时候，

就会导致消息重复发送（失去了幂等性）。而事务可以弥补这个缺憾，事务可以保证对多个分区写入操作的原子性。操作的原子性是指多个操作要么全部成功，要么全部失败，不存在

部分成功部分失败的可能。为了实现事务，应用程序必须提供唯一的`transactionalId`，这个参数通过客户端程序来进行设定。如下所示：

```
1 | spring.kafka.producer.transaction-id-prefix=order_tx.    # 表示开启事务机制
```

并且事务机制的使用需要幂等性的支持，所以我们还需要开启幂等性：`enable.idempotence = true`。如果没有开启幂等性的支持，就会报错，如下所示：

```
1 | Caused by: org.apache.kafka.common.config.ConfigException: Cannot set a transactional.id without also enabling idempotence.
```

事务机制实现的两种方式：

1、第一种方式

```
1 | // 事务消息发送的第一种方式
2 | public static void sendTransactionMessageMethod01() {
```

```

3
4 // 发送事务消息
5 kafkaTemplate.executeInTransaction((operations) -> {
6
7 // 发送消息
8 kafkaTemplate.send("itcast" , 0 , "order" , "事务消息 ----> 1")
9 ;
10 kafkaTemplate.send("itcast" , 0 , "order" , "事务消息 ----> 2")
11 ;
12 // 产生异常代码
13 int a = 1 / 0 ;
14 kafkaTemplate.send("itcast" , 0 , "order" , "事务消息 ----> 3")
15 ;
16 // 返回true, 表示发送成功
17 return true ;
18
19 }) ;
20
21 LOGGER.info("send transaction message to local cache success ");
22 }

```

2、第二种方式

```

1 @Transactional(rollbackFor = RuntimeException.class)
2 public void sendTransactionMessage() {
3
4 // 发送消息
5 kafkaTemplate.send("itcast" , 0 , "order" , "事务消息 ----> 1") ;
6 kafkaTemplate.send("itcast" , 0 , "order" , "事务消息 ----> 2") ;
7
8 // 产生异常代码
9 int a = 1 / 0 ;
10
11 kafkaTemplate.send("itcast" , 0 , "order" , "事务消息 ----> 3") ;
12
13 }

```

3.4.2 消费端消息重复

1、根本原因

数据消费完没有及时提交offset到broker。

解决方案

1、取消自动提交

每次消费完或者程序退出时手动提交。

2、下游做幂等

一般的解决方案是让下游做幂等。

3.5 Kafka消息队列高可用

一般情况下，在使用消息队列的时候，我们都需要保证消息队列的高可用。要保证消息队列的高可用，那么我们就需要搭建消息队列集群。

3.5.1 集群搭建

zookeeper集群搭建

真实的集群是需要部署在不同的服务器上的，但是在我们的测试时同时启动几个虚拟机内存会吃不消，所以这里我们搭建伪集群，也就是把所有的服务都搭建在一台虚拟机上，用端口进行区分。

我们这里要求搭建一个三个节点的Zookeeper集群（伪集群）。

具体的实现步骤如下所示：

- 1、为了不影响集群的搭建，我们可以删除原有的zookeeper和Kafka容器，然后我们重新进行创建。
- 2、创建3个zookeeper容器

```
1 docker run -di --network=docker-network --ip=172.19.0.61 --name=zk_01 -p 2181:2181 --privileged=true wurstmeister/zookeeper /bin/bash
2 docker run -di --network=docker-network --ip=172.19.0.62 --name=zk_02 -p 2182:2181 --privileged=true wurstmeister/zookeeper /bin/bash
3 docker run -di --network=docker-network --ip=172.19.0.63 --name=zk_03 -p 2183:2181 --privileged=true wurstmeister/zookeeper /bin/bash
```

- 3、修改zookeeper的配置文件，添加如下内容

```
1 server.0=172.19.0.61:2888:3888
2 server.1=172.19.0.62:2888:3888
3 server.2=172.19.0.63:2888:3888
```

docker下安装vi，按方向键变成了输入A B C D,解决方案：

```
1 echo "set nocompatible" >> ~/.vimrc
2 echo "set backspace=indent,eol,start" >> ~/.vimrc
3 echo "set fileencodings=utf-8,ucs-bom,gb18030,gbk,gb2312,cp936" >> ~/.vimrc
4 echo "set termencoding=utf-8" >> ~/.vimrc
5 echo "set encoding=utf-8" >> ~/.vimrc
```

- 4、在每一个容器的data目录下创建一个myid文件，然后在每一个myid文件中指定当前server的id
- 5、启动zookeeper，查看节点状态

Kafka集群搭建

Kafka集群的搭建步骤如下所示：

1、创建3个Kafka实例

```
1 docker run -di --network=host --name=kafka_01 -v
  /etc/localtime:/etc/localtime --privileged=true
  wurstmeister/kafka:latest /bin/bash
2 docker run -di --network=host --name=kafka_02 -v
  /etc/localtime:/etc/localtime --privileged=true
  wurstmeister/kafka:latest /bin/bash
3 docker run -di --network=host --name=kafka_03 -v
  /etc/localtime:/etc/localtime --privileged=true
  wurstmeister/kafka:latest /bin/bash
```

2、更改配置文件

```
1 broker.id=0 # 表示broker的编号，如
   果集群中有多个broker，则每个broker的编号需要设置的不同
2 port=9092 # 端口号
3 listeners=PLAINTEXT://192.168.23.131:9092 # broker对外提供的服
   务入口地址
4 log.dirs=/tmp/kafka-logs # 设置存放消息日志文件
   的地址
5 zookeeper.connect=172.19.0.61:2181,172.19.0.62:2181,172.19.0.63:2181
   # Kafka所需Zookeeper集群地址，教学中Zookeeper和Kafka都安装本机
```

3、启动kafka服务

3.5.3 Kafka集群测试

接下来我们就对我们的集群进行测试：

```
bin/kafka-topics.sh --zookeeper 172.19.0.61:2181 --create --topic itcast --partitions 2 --replication-factor 3 # 创建主题
bin/kafka-topics.sh --zookeeper 172.19.0.61:2181 --describe --topic itcast # 查看主题详情信息
bin/kafka-console-consumer.sh --bootstrap-server 192.168.23.131:9092 --topic itcast # 开启一个消费者
bin/kafka-console-producer.sh --broker-list 192.168.23.131:9092 --topic itcast # 开启一个生产者
```

3.5.4 Kafka集群监控

为了方便的对Kafka进行操作，我们可以使用Kafka的一个集群监控工具Kafka Eagle。

1、Kafka Eagle下载与安装

- [[Kafka Eagle 下载地址](#)]
- [[Kafka Eagle 使用文档](#)]

2、修改配置文件：/root/kafka-eagle-web-1.4.8/conf/system-config.properties

```
1 #####
2 # multi zookeeper & kafka cluster list
3 #####
4 kafka.eagle.zk.cluster.alias=cluster1
5 cluster1.zk.list=172.19.0.61:2181,172.19.0.62:2181,172.19.0.63:2181
6 # cluster2.zk.list=xdn10:2181,xdn11:2181,xdn12:2181
7
8 #####
9 # broker size online list
10 #####
```

```
11 cluster1.kafka.eagle.broker.size=20
12
13 #####
14 # zk client thread limit
15 #####
16 kafka.zk.limit.size=25
17
18 #####
19 # kafka eagle webui port
20 #####
21 kafka.eagle.webui.port=8090
22
23 #####
24 # kafka offset storage
25 #####
26 cluster1.kafka.eagle.offset.storage=kafka
27 # cluster2.kafka.eagle.offset.storage=zk
28
29 #####
30 # kafka metrics, 15 days by default
31 #####
32 kafka.eagle.metrics.charts=false
33 kafka.eagle.metrics.retain=15
34
35
36 #####
37 # kafka sql topic records max
38 #####
39 kafka.eagle.sql.topic.records.max=5000
40 kafka.eagle.sql.fix.error=false
41
42 #####
43 # delete kafka topic token
44 #####
45 kafka.eagle.topic.token=keadmin
46
47 #####
48 # kafka sasl authenticate
49 #####
50 #cluster1.kafka.eagle.sasl.enable=false
51 #cluster1.kafka.eagle.sasl.protocol=SASL_PLAINTEXT
52 #cluster1.kafka.eagle.sasl.mechanism=SCRAM-SHA-256
```

```
53 #cluster1.kafka.eagle.sasl.jaas.config=org.apache.kafka.common.security
    .scram.ScramLoginModule required username="kafka" password="kafka-
    eagle";
54 #cluster1.kafka.eagle.sasl.client.id=
55 #cluster1.kafka.eagle.sasl.cgroup.enable=false
56 #cluster1.kafka.eagle.sasl.cgroup.topics=
57
58 #####
59 # kafka sqlite jdbc driver address
60 #####
61 kafka.eagle.driver=org.sqlite.JDBC
62 kafka.eagle.url=jdbc:sqlite:/root/kafka-eagle-web-1.4.8/db/ke.db
63 kafka.eagle.username=root
64 kafka.eagle.password=123456
```

3、启动： ./ke.sh start

4、登录： <http://192.168.23.131:8090/ke> 访问，使用admin/123456进行登录

第一次访问的时候，比较慢，因此需要从zookeeper中获取kafka的监控数据。

ID	Topic Name	Partitions	Broker Spread	Broker Skewed	Broker Leader Skewed	Created	Modify	Operate
1	haha	3	100%	0%	0%	2020-06-21 22:00:17	2020-06-21 22:00:17	Action
2	l1cast	2	100%	0%	33%	2020-06-21 04:56:32	2020-06-21 04:56:32	Action
3	transaction_state	50	100%	0%	0%	2020-06-21 22:34:24	2020-06-21 22:34:24	Action
4	theima	1	33%	0%	0%	2020-06-21 05:09:47	2020-06-21 05:09:47	Action

3.6 Kafka为什么快

Kafka消息被持久化到本地磁盘。Rabbitmq存储消息的时候使用的是内存和磁盘的方式进行存储；按照我们的理解Rabbitmq的消息吞吐量应该大于Kafka的消息吞吐量，但是相反

Rabbitmq的消息吞吐量反而小于Kafka的，那么为什么呢？

3.6.1 分区管理

Kafka可以将主题划分为多个分区（Partition），会根据分区规则选择把消息存储到哪个分区中，只要分区规则设置的合理，那么所有的消息将会被均匀的分布到不同的分区中，这样

就实现了负载均衡和水平扩展。另外，多个订阅者可以从一个或者多个分区中同时消费数据，以支撑海量数据处理能力。顺便说一句，由于**消息是以追加的方法存储到分区中的，多个**

分区顺序写磁盘的总效率要比随机写内存还要高（引用Apache Kafka – A High Throughput Distributed Messaging System的观点），是Kafka高吞吐率的重要保证之一。

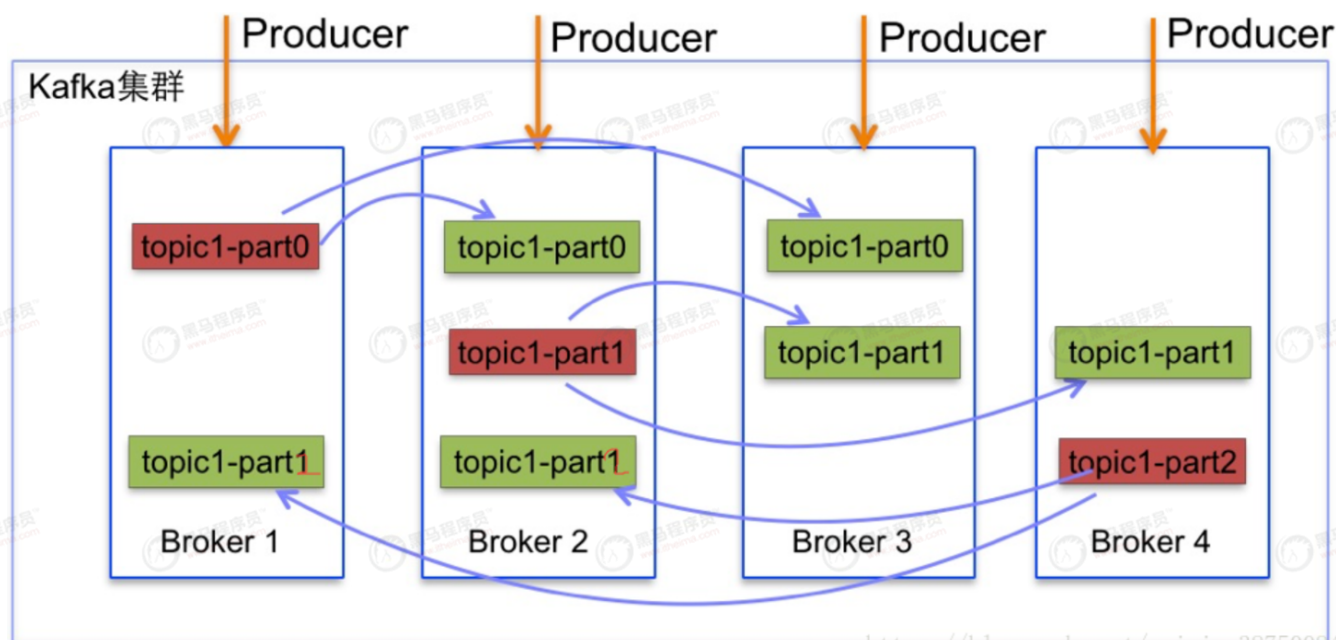
分区副本机制

由于Producer和Consumer都只会与Leader角色的分区副本相连，所以kafka需要以集群的组织形式提供主题下的消息高可用。kafka支持主备复制，所以消息具备高可用和持久性。

一个分区可以有多个副本，这些副本保存在不同的broker上。每个分区的副本中都会有一个作为Leader。当一个broker失败时，Leader在这台broker上的分区都会变得不可用，kafka

会自动移除Leader，再其他副本中选一个作为新的Leader。在通常情况下，增加分区可以提供kafka集群的吞吐量。然而，也应该意识到集群的总分区数或是单台服务器上的分区数过

多，会增加不可用及延迟的风险。



可以通过如下命令查看某一个主题的分区情况：

```
1 bin/kafka-topics.sh --zookeeper 172.19.0.60:2181 --describe --topic itheima
2 Topic: itheima PartitionCount: 3 ReplicationFactor: 1 Configs:
3 Topic: itheima Partition: 0 Leader: 0 Replicas: 0 Isr: 0
4 Topic: itheima Partition: 1 Leader: 0 Replicas: 0 Isr: 0
5 Topic: itheima Partition: 2 Leader: 0 Replicas: 0 Isr: 0
```

Leader和Replicas以及Isr中的数字表示的是broker的标号

分区leader选举

可以预见的是,如果某个分区的Leader挂了,那么其它跟随者将会进行选举产生一个新的leader,之后所有的读写就会转移到这个新的Leader上,在kafka中,其不是采用常见的多数选举的

方式进行副本的Leader选举,而是会在Zookeeper上针对每个Topic维护一个称为ISR (in-sync replica, 已同步的副本) 的集合,显然还有一些副本没有来得及同步。只有这个ISR列表里

面的才有资格成为leader(先使用ISR里面的第一个, 如果不行依次类推, 因为ISR里面的是同步副本, 消息是最完整且各个节点都是一样的)。通过ISR,kafka可以容忍的失败数比较高。

假设某个topic有 $f+1$ 个副本, kafka可以容忍 f 个不可用,当然如果全部ISR里面的副本都不可用,也可以选择其他可用的副本,只是存在数据的不一致。

分区重新分配

我们往已经部署好的Kafka集群里面添加机器是最正常不过的需求，而且添加起来非常地方便，我们需要做的事是从已经部署好的Kafka节点中复制相应的配置文件，然后把里面的

broker id修改成全局唯一的，最后启动这个节点即可将它加入到现有Kafka集群中。但是问题来了，新添加的Kafka节点并不会自动地分配数据，所以无法分担集群的负载，除非我

们新建一个topic。但是现在我们想手动将部分分区移到新添加的Kafka节点上，Kafka内部提供了相关的工具来重新分布某个topic的分区。

具体的实现步骤如下所示：

1、比如某一个主题的分区信息如下所示：

Topic Meta Info							
Topic	Partition	Log Size	Leader	Replicas	In Sync Replicas	Preferred Leader	Under Replicated
itheima	0	0	1	[1, 0, 2]	[1,0,2]	true	false
itheima	1	0	0	[0, 2, 1]	[0,2,1]	true	false
itheima	2	0	2	[2, 1, 0]	[2,1,0]	true	false

Showing 1 to 3 of 3 entries

Previous 1 Next

2、给某一个分区在添加一个新的分区

```
1 bin/kafka-topics.sh --alter --zookeeper 172.19.0.61:2181 --topic itcast --partitions 4
```

添加完毕以后，分区的信息如下所示：

Topic Meta Info							
Topic	Partition	Log Size	Leader	Replicas	In Sync Replicas	Preferred Leader	Under Replicated
itheima	0	0	1	[1, 0, 2]	[1,0,2]	true	false
itheima	1	0	0	[0, 2, 1]	[0,2,1]	true	false
itheima	2	0	2	[2, 1, 0]	[2,1,0]	true	false
itheima	3	0	1	[1, 2, 0]	[1,2,0]	true	false

Showing 1 to 4 of 4 entries

Previous 1 Next

这样会导致3个Broker上有重新维护了更多的分区节点。

3、再次创建一个kafka的容器

```
1 docker run -di --network=host --name=kafka_04 -v /etc/localtime:/etc/localtime --privileged=true wurstmeister/kafka:latest /bin/bash
```

查看itheima主题的分区情况，如下所示：

Topic Meta Info							
Topic	Partition	Log Size	Leader	Replicas	In Sync Replicas	Preferred Leader	Under Replicated
itheima	0	0	1	[1, 0, 2]	[1,0,2]	true	false
itheima	1	0	0	[0, 2, 1]	[0,2,1]	true	false
itheima	2	0	2	[2, 1, 0]	[2,1,0]	true	false
itheima	3	0	1	[1, 2, 0]	[1,2,0]	true	false

Showing 1 to 4 of 4 entries

Previous 1 Next

和之前没有任何的变化。

3、修改集群配置文件

```
1 broker.id=3 # 表示broker的编号, 如果集群中有多个broker, 则每个broker的编号需要设置的不同
2 port=9095 # 端口号
3 listeners=PLAINTEXT://192.168.23.131:9095 # broker对外提供的服务入口地址
4 log.dirs=/tmp/kafka-logs # 设置存放消息日志文件的地址
5 zookeeper.connect=172.19.0.61:2181,172.19.0.62:2181,172.19.0.63:2181
# kafka所需Zookeeper集群地址, 教学中zookeeper和Kafka都安装本机
```

4、重新分配

现在我们需要将原先分布在broker 1-3节点上的分区重新分布到broker 1-4节点上, 借助kafka-reassign-partitions.sh工具生成reassign plan, 不过我们先得按照要求定义一个文件, 里面

说明哪些topic需要重新分区, 文件内容如下:

```
1 itcast@Server-node:/mnt/d/kafka-cluster/kafka-1$ cat reassign.json
2 {"topics":[{"topic":"itcast"}],
3  "version":1
4  }
```

然后使用 kafka-reassign-partitions.sh 工具生成reassign plan

```
bin/kafka-reassign-partitions.sh --zookeeper 172.19.0.61:2181 --topics-to-move-json-file
reassign.json --broker-list "0,1,2,3" --generate
```

命令会输出两个字符串,如下所示:

```

1 Current partition replica assignment
2 {"version":1,"partitions":[{"topic":"itheima","partition":2,"replicas":
  [2,1,0],"log_dirs":["any","any","any"]},
  {"topic":"itheima","partition":1,"replicas":[0,2,1],"log_dirs":
  ["any","any","any"]},{"topic":"itheima","partition":0,"replicas":
  [1,0,2],"log_dirs":["any","any","any"]},
  {"topic":"itheima","partition":3,"replicas":[1,2,0],"log_dirs":
  ["any","any","any"]}]}
3
4 Proposed partition reassignment configuration
5 {"version":1,"partitions":[{"topic":"itheima","partition":2,"replicas":
  [3,1,2],"log_dirs":["any","any","any"]},
  {"topic":"itheima","partition":1,"replicas":[2,0,1],"log_dirs":
  ["any","any","any"]},{"topic":"itheima","partition":3,"replicas":
  [0,2,3],"log_dirs":["any","any","any"]},
  {"topic":"itheima","partition":0,"replicas":[1,3,0],"log_dirs":
  ["any","any","any"]}]}

```

第一个JSON内容为当前的分区副本分配情况，第二个为重新分配的候选方案，注意这里只是生成一份可行性的方案，并没有真正执行重分配的动作。

我们将第二个JSON内容保存到名为result.json文件里面（文件名不重要，文件格式也不一定要以json为结尾，只要保证内容是json即可），然后执行这些reassign plan。如下所示：

执行分配策略：

```

1 bin/kafka-reassign-partitions.sh --zookeeper 172.19.0.61:2181 --
  reassignment-json-file result.json --execute

```

执行完毕以后，看出分区信息：

Topic	Partition	LogSize	Leader	Replicas	In Sync Replicas	Preferred Leader	Under Replicated
itheima	0	0	1	[1, 3, 0]	[1,0,3]	true	false
itheima	1	0	2	[2, 0, 1]	[0,2,1]	true	false
itheima	2	0	3	[3, 1, 2]	[2,1,3]	true	false
itheima	3	0	0	[0, 2, 3]	[2,0,3]	true	false

Showing 1 to 4 of 4 entries

Previous 1 Next

修改副本因子

修改副本因子，其实就是修改分区的副本数。

场景

修改副本因子的使用场景也很多，比如在创建主题时填写了错误的副本因子数而需要修改，再比如运行一段时间之后想要通过增加副本因子数来提高容错性和可靠性。修改副本因此也

是通过kafka-reassign-partitions.sh脚本实现的。仔细观察我们刚才对itheima进行分区重新分配以后的结果：

```
1 {"version":1,"partitions":[{"topic":"itheima","partition":2,"replicas":
  [3,1,2],"log_dirs":["any","any","any"]},
  {"topic":"itheima","partition":1,"replicas":[2,0,1],"log_dirs":
  ["any","any","any"]},{"topic":"itheima","partition":3,"replicas":
  [0,2,3],"log_dirs":["any","any","any"]},
  {"topic":"itheima","partition":0,"replicas":[1,3,0],"log_dirs":
  ["any","any","any"]}]]}
```

通过观察JSON内容里的replicas都是3个副本。我们可以更改指定分区数的副本，具体的实现如下所示：

1、创建一个json文件，定义指定分区的副本数据

```

1 {
2   "version":1,
3   "partitions":[
4     {"topic":"itcast","partition":2,"replicas":[0,1]}
5   ]
6 }

```

2、然后执行脚本文件

```

1 bash-4.4# bin/kafka-reassign-partitions.sh --zookeeper 172.19.0.61:2181
--reassignment-json-file replication-factor.json --execute
2 Current partition replica assignment
3
4 {"version":1,"partitions":[{"topic":"itheima","partition":2,"replicas":
[3,1,2],"log_dirs":["any","any","any"]},
{"topic":"itheima","partition":1,"replicas":[2,0,1],"log_dirs":
["any","any","any"]}, {"topic":"itheima","partition":0,"replicas":
[1,3,0],"log_dirs":["any","any","any"]},
{"topic":"itheima","partition":3,"replicas":[0,2,3],"log_dirs":
["any","any","any"]}]}
5
6 Save this to use as the --reassignment-json-file option during rollback
7 Successfully started reassignment of partitions.

```

执行完毕以后，我们再次看出itheima的分区副本情况，如下所示：

Topic Meta Info							
Topic	Partition	LogSize	Leader	Replicas	In Sync Replicas	Preferred Leader	Under Replicated
itheima	0	0	0	[0, 1]	[1,0]	true	false
itheima	1	0	2	[2, 0, 1]	[0,2,1]	true	false
itheima	2	0	3	[3, 1, 2]	[2,1,3]	true	false
itheima	3	0	0	[0, 2, 3]	[2,0,3]	true	false

Showing 1 to 4 of 4 entries

Previous 1 Next

3.6.2 kafka存储

Kafka作为一个支持大数据量写入写出的消息队列，由于是基于Scala和Java实现的，而Scala和Java均需要在JVM上运行，所以如果是基于内存的方式，即JVM的堆来进行数据存储则需要开

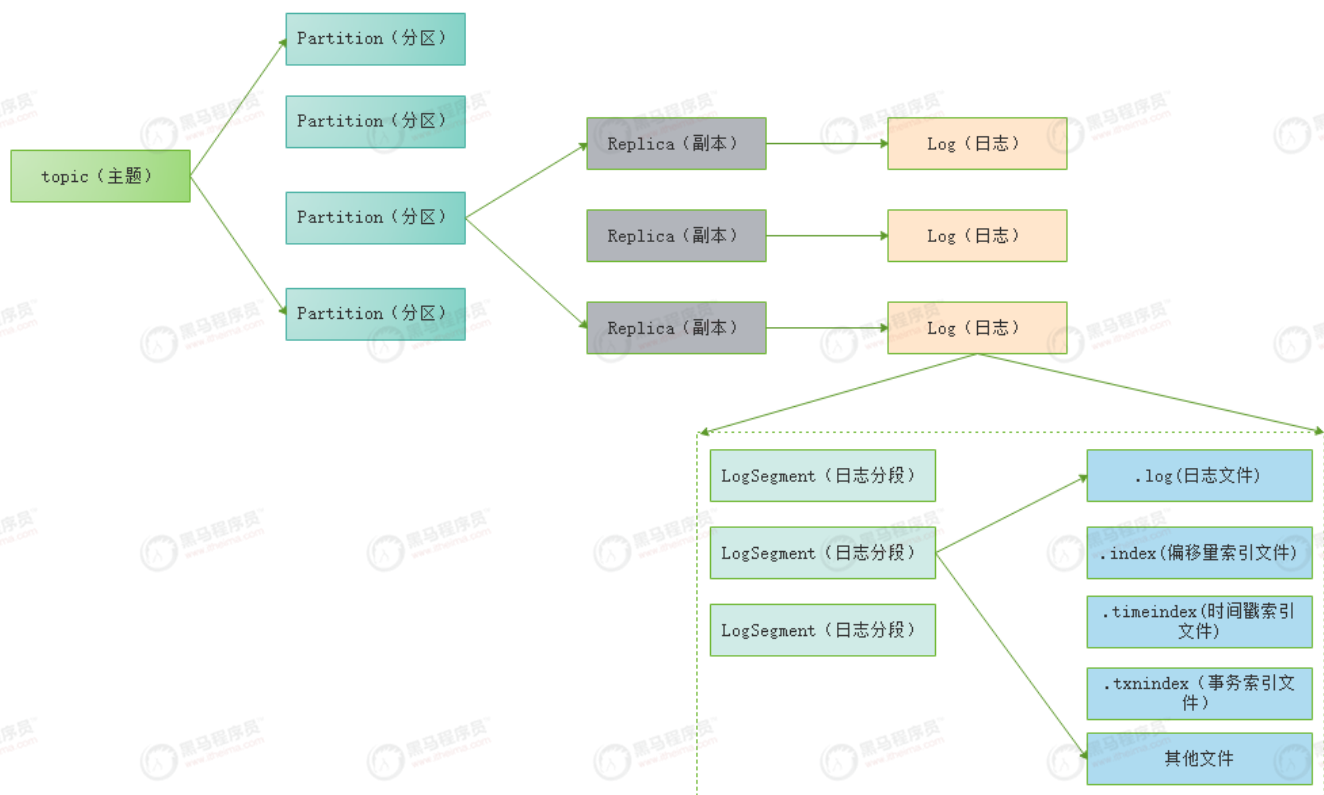
辟很大的堆来支持数据读写，从而会导致GC频繁影响性能。考虑到这些因素，kafka是使用磁盘而不是kafka服务器broker进程内存来进行数据存储，并且基于磁盘顺序读写和MMAP技术来实现高性能。

存储结构介绍

Kafka一个topic下可以存在很多个分区，不考虑分区副本的情况下。一个分区对应一个日志（Log）。为了防止Log过大，Kafka又引入了日志分段（LogSegment）的概念，将Log切分为

多个LogSegment，相当于一个巨型文件被平均分配为多个相对较小的文件，这样也便于消息的维护和清理。事实上，Log和LogSegment对应于磁盘上的一个日志文件和两个索引文件，

以及可能的其他文件（比如：以".txnindex"为后缀的事务索引文件）。如下图所示：



Log对应了一个命名形式为-的文件夹。举个例子，假设有一个名为"itheima"的主题，此主题中具有4个分区，那么在实际的物理存储上表现为"itheima-0", "itheima-

1", "itheima-2", "itheima-3"这4个文件夹 (/tmp/kafka-logs)。

```

drwxr-xr-x 2 root root      141 Jun 22 03:44 transaction-state-1
-rw-r--r-- 1 root root         4 Jun 22 02:11 cleaner-offset-checkpoint
drwxr-xr-x 2 root root      178 Jun 22 03:49 itheima-0
drwxr-xr-x 2 root root      141 Jun 22 02:11 itheima-1
drwxr-xr-x 2 root root      141 Jun 22 02:11 itheima-2
drwxr-xr-x 2 root root      141 Jun 22 02:11 itheima-3
  
```

向Log中追加消息时是**顺序写入**的，只有最后一个LogSegment才能执行写入操作，在此之前所有的LogSegment都不能写入数据。为了方便描述，我们将最后一个LogSegment为"activeSegment"，

即表示当前活跃的日志分段。每当segment文件达到一定的大小，则会创建一个新的segment文件（具体大小在server.properties中通过配置log.segment.bytes=1073741824：默认为1G）。之后

追加的消息将写入新的activeSegment。文件名是以该文件的第一个数据相对于该分区的全局offset命名的。名称固定为20位数字，没有达到的位数则用0填充。比如第一个LogSegment的基准偏移

量为0，对应的日志文件的名称为00000000000000000000.log。

```

drwxr-xr-x 2 root root      178 Jun 22 03:49
drwxr-xr-x 33 root root     4096 Jun 22 03:16
-rw-r--r-- 1 root root 10485760 Jun 22 03:49 00000000000000000000.index
-rw-r--r-- 1 root root 75221542 Jun 22 03:49 00000000000000000000.log
-rw-r--r-- 1 root root 10485736 Jun 22 03:49 00000000000000000000.timeindex
-rw-r--r-- 1 root root      34 Jun 22 03:49 00000000000000000000.txnindex
-rw-r--r-- 1 root root       8 Jun 22 03:44 leader-epoch-checkpoint
  
```

消息写入

存储器的两种输入输出方式：

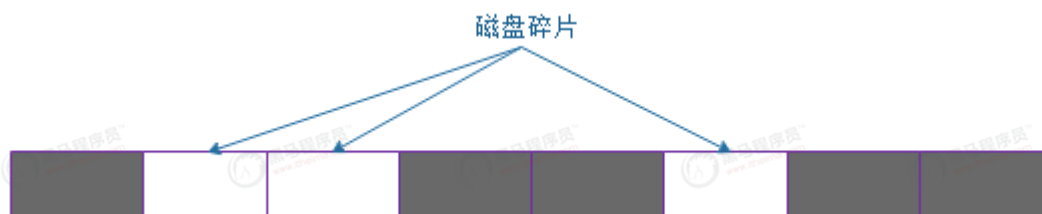
- 1、随机读写
- 2、顺序读写

随机读写：存储的数据在磁盘中占据空间，对于一个新磁盘，操作系统会将数据文件依次写入磁盘，当有些数据被删除时，就会空出该数据原来占有的存储空间，时间长了，不断的写入、

删除数据，就会产生很多零零散散的存储空间，就会造成一个较大的数据文件放在许多不连续的存贮空间上，读写这这部分数据时，就是随机读写，磁头要不断的调整磁道的位置，以在

不同位置上的读写数据，相对于连续空间上的顺序读写，要耗时很多。

如下图所示：当删除某些数据的时候，就会产生很多不连续的磁盘存储空间（磁盘碎片）



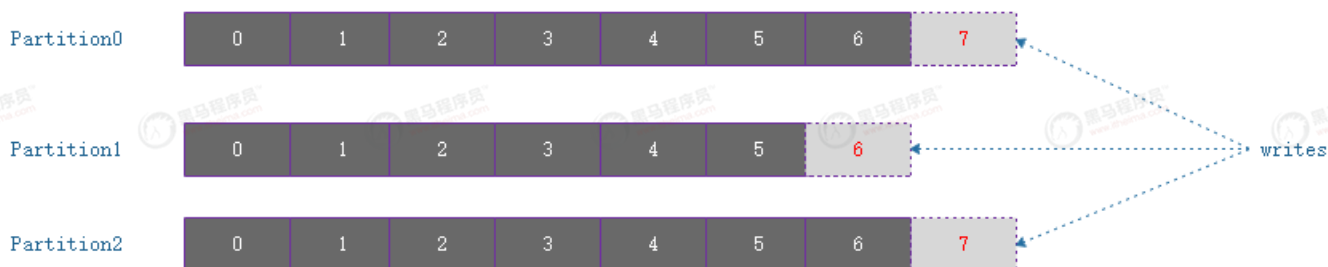
新文件的存储：

新存储的文件



顺序读写：就是在文件的末尾以追加的方式来写入消息，在Kafka中只能在日志文件的末尾追加新的消息，并且也不允许修改和删除已写入的消息。

如下图所示：



这种方法有一个缺陷——没有办法删除数据，所以Kafka是不会删除数据的，它会把所有的数据都保留下来，消费者在消费消息的时候是通过"消费位移"（offset）来决定从哪个位置开始进行

消息的消费。

页缓存

顺序写入是Kafka高吞吐量的一个原因，当然即使采用的是磁盘的顺序写入，那么也是没有办法和内存相比的。因为为了再一次提高Kakfa的吞吐量，Kafka采用了Memory Mapped Files

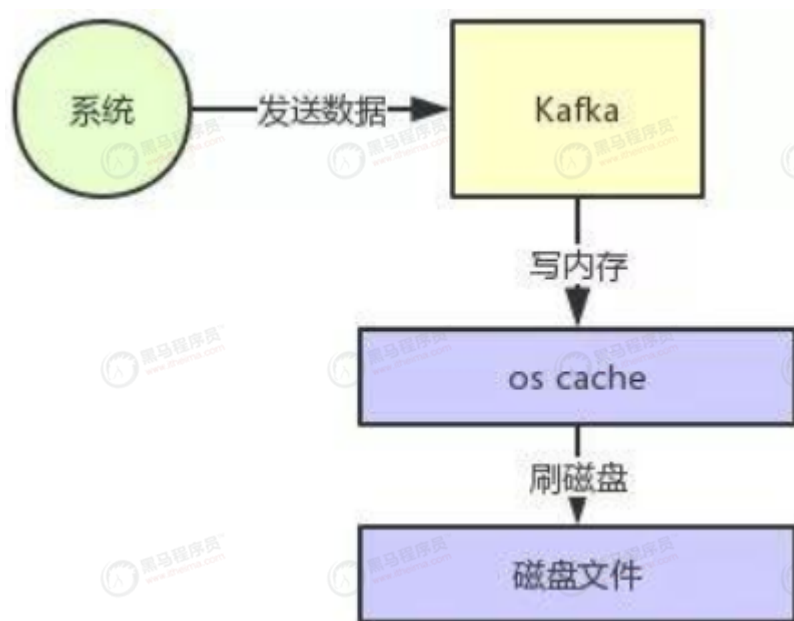
(后面简称 mmap)也被翻译成内存映射文件，它的工作原理是直接利用操作系统的 page cache 来实现文件到物理内存的直接映射，完成映射之后你对物理内存的操作会被同步到硬盘上

(操作系统在适当的时候)。

操作系统本身有一层缓存，叫做page cache，是在内存里的缓存，我们也可以称之为os cache，意思就是操作系统自己管理的缓存。你在写入磁盘文件的时候，可以直接写入这个

os cache里，也就是仅仅写入内存中，接下来由操作系统自己决定什么时候把os cache里的数据真的刷入磁盘文件中（每5秒检查一次是否需要将页缓存数据同步到磁盘文件）。仅仅

这一个步骤，就可以将磁盘文件写性能提升很多了，因为其实这里相当于是在写内存，不是在写磁盘，大家看下图：



所以大家就知道了，上面那个图里，Kafka在写数据的时候，一方面基于了os层面的page cache来写数据，所以性能很高，本质就是在写内存罢了。另外一个，他是采用磁盘顺序写的方式，所以即使数据刷入磁盘的时候，性能也是极高的，也跟写内存是差不多的。基于上面两点，kafka就实现了写入数据的超高性能。

日志清理

Kafka将消息存储在磁盘中，为了控制磁盘占用空间的不断增加需要对消息做一定的清理操作。Kafka中每一个分区副本都对应了一个Log，而Log又可以分为多个日志分段，这样也便于日志

的清理操作。Kafka提供了两种日志清理的策略：

- 1、日志删除（Log Retention）：按照一定的保留策略直接删除不符合条件的日志分段。
- 2、日志压缩（Log Compaction）：针对每个消息的Key进行整合，对于有相同key的不同value值，只保留最后一个版本。

我们可以通过broker端参数：**log.cleanup.policy**来设置日志清理策略，此参数的默认值为"delete"（可以在kafka启动时打印的日志中查看到），即采用日志删除的清理策略。如果要采用日

志压缩的清理策略，就需要将log.cleanup.policy设置为"compact",并且还需要将**log.cleaner.enable**的值设置为true。通过将log.cleanup.policy参数设置为"delete,compact",还可以同时支

持日志删除和日志压缩两种策略。最常见的日志清理操作是日志删除。

在Kafka的日志管理器中会有一个专门的日志删除任务来周期性地检测和删除不符合保留条件的日志分段文件，这个周期可以通过broker端参数`log.retention.check.interval.ms`来配置，默认

值为300000，即5分钟。当前日志分段的常见的保留策略有2种：

- 1、基于时间的保留策略
- 2、基于日志大小的保留策略

基于时间：

日志删除任务会检查当前日志文件中是否有保留时间超过设定的阈值来寻找可删除的日志分段文件集合。日志分段保留时间的阈值可以通过如下参数进行设置：

```
1 | log.retention.hours = 168           # 单位为小时，默认的取值为7天
    优先级最低
2 | log.retention.minutes=null          # 单位为分钟
    优先级次之
3 | log.retention.ms=null              # 单位为毫秒
    优先级最高
```

删除日志分段文件时，首先会从Log对象中所维护日志分段的跳跃表中移除待删除的日志分段，以保证没有线程对这些日志分段进行读取操作。然后将日志分段所对应的所有文件添加上".delete"

的后缀（当然也包含对应的索引文件）。最后交由一个以"delete-file"的命令的延迟任务来删除这些以".delete"为后缀的文件，这个任务的延迟时间可以通过`file.delete.delay.ms`参数来调配，此参数的默认值为60000，即1分钟。

基于大小：

日志删除任务会检查当前日志的大小是否超过设定的阈值来查找可删除的日志分段的文件集合，可以通过如下参数来设定日志大小的阈值：

```
1 | log.retention.bytes = -1 # 默认为-1，表示无穷大
```

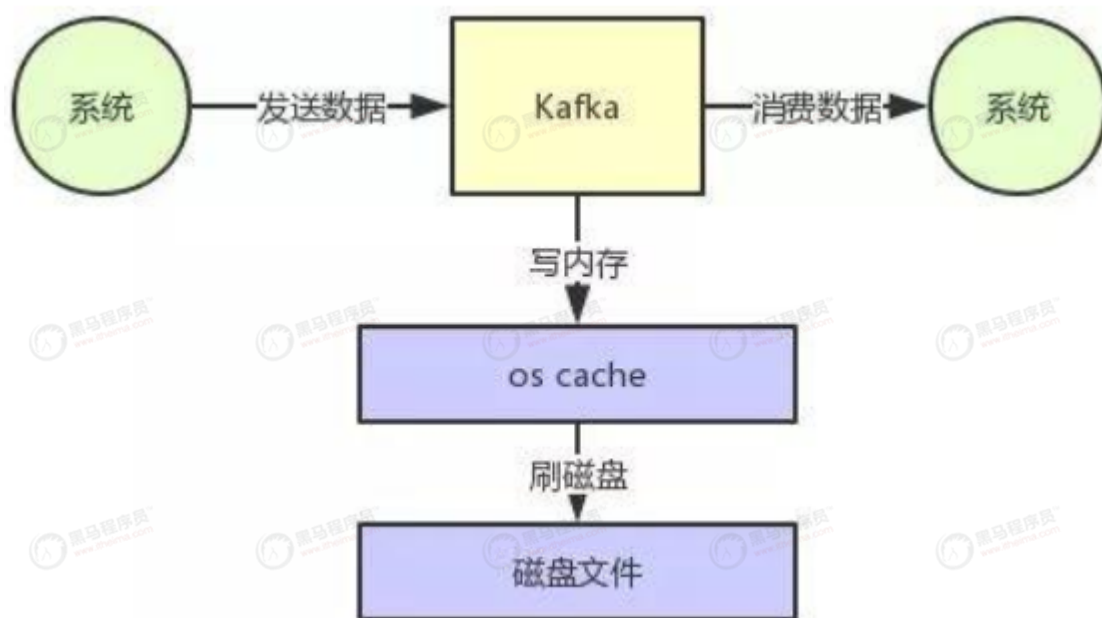
注：该参数配置的是Log中所有日志文件的总大小，而不是单个日志分段的大小（.log日志文件）

零拷贝技术

说完了写入这块，再来谈谈消费这块。

大家应该都知道，从Kafka里我们经常要消费数据，那么消费的时候实际上就是要从kafka的磁盘文件里读取某条数据然后发送给下游的消费者，如下图所示。

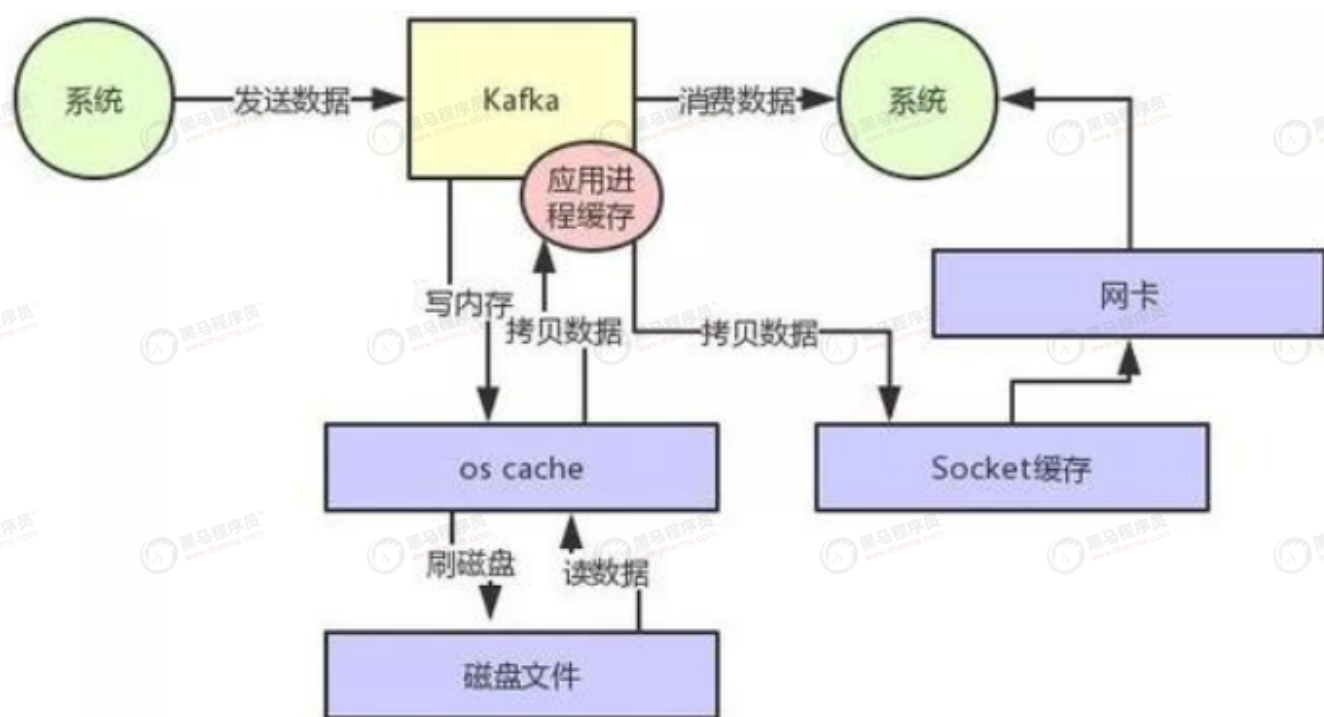
那么这里如果频繁的从磁盘读数数据然后发给消费者，性能瓶颈在哪里呢？



假设要是kafka什么优化都不做，就是很简单的从磁盘读数据发送给下游的消费者，那么大概过程如下所示：

- 1、先看看要读的数据在不在os cache里，如果不在的话就从磁盘文件里读取数据后放入os cache。
- 2、接着从操作系统的os cache里拷贝数据到应用程序进程的缓存里。
- 3、再从应用程序进程的缓存里拷贝数据到操作系统层面的Socket缓存里。
- 4、最后从Socket缓存里提取数据后发送到网卡，最后发送出去给下游消费。

整个过程，如下图所示：



完成上述的操作，进行了两次copy：

- 1、一次是从操作系统的cache里拷贝到应用进程的缓存里。

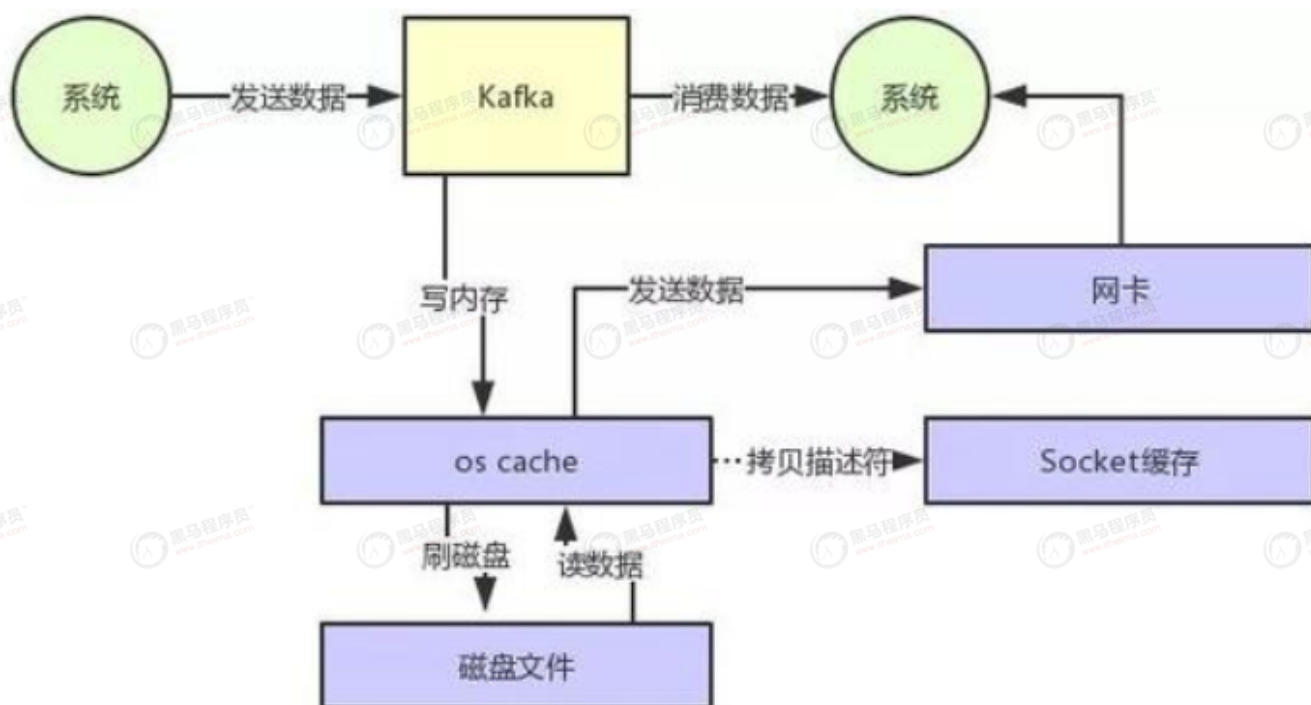
2、从应用程序缓存里拷贝回操作系统的Socket缓存里。

而且为了进行这两次拷贝，中间还发生了好几次上下文切换，一会儿是应用程序在执行，一会儿上下文切换到操作系统来执行。所以这种方式来读取数据是比较消耗性能的。

Kafka为了解决这个问题，在读数据的时候是引入**零拷贝技术**。

也就是说，直接让操作系统的cache中的数据发送到网卡后传输给下游的消费者，中间跳过了两次拷贝数据的步骤，Socket缓存中仅仅会拷贝一个描述符过去，不会拷贝数据到Socket缓存。

大家看下图，体会一下这个精妙的过程：



通过零拷贝技术，就不需要把os cache里的数据拷贝到应用缓存，再从应用缓存拷贝到Socket缓存了，两次拷贝都省略了，所以叫做零拷贝。

对Socket缓存仅仅就是拷贝**数据的描述符过去**，然后数据就直接从os cache中发送到网卡上去了，这个过程大大的提升了数据消费时读取文件数据的性能。

4 第四章：总结

本课程是以电商的秒杀场景作为切入点，来深入讲解一下MQ体系中的Rabbitmq和Kafka。讲解了一下在秒杀系统中在使用Rabbitmq或者Kafka的时候如何保证秒杀的公平性和不超卖。

秒杀公平性MQ的保证：就是就是要保证消息的可靠性传输和顺序性以及MQ的高可用

秒杀不超卖MQ的保证：防止消息重复消费，下游业务做幂等性处理

