



## 图文 118 Consumer是如何从Broker上拉取一批消息过来处理的?

29 人次阅读 2020-04-10 07:00:00

[返回](#)  
[前进](#)  
[重新加载](#)  
[打印](#)
[详情](#) [评论](#)


狸猫技术

## Consumer是如何从Broker上拉取一批消息过来处理的?

进店逛

## 相关频道



从 0 开始  
间件实战  
已更新1



继《从零开始带你成为JVM实战高手》后，阿里资深技术专家携新作再度出山，重磅推荐：

(点击下方蓝字试听)

[《从零开始带你成为MySQL实战优化高手》](#)

今天要讲的内容就是本专栏的最后一篇内容了，这篇内容我寻思良久，觉得其实并没有必要重新写，因为之前的文章里有一篇已经把这个消费的机制讲解的很清楚了，这里权且当做复习，放在最后一篇文章，大家再看一次，同时在最后我们将会对消费这块的源码做一点提示，大家可以自己去慢慢看。

**1、消费组到底是个什么概念？**

在理解了Broker数据存储机制以及Broker高可用主从同步机制之后，我们就可以来看一下消费者是如何从Broker获取消息，并且进行处理以及维护消费进度的。首先，我们需要了解的第一个概念，就是消费者组。

消费者组的意思，就是让你给一组消费者起一个名字。比如说我们有一个Topic叫做“TopicOrderPaySuccess”，那么假设有库存系统、积分系统、营销系统、仓储系统他们都要去消费这个Topic中的数据。

此时我们应该给那四个系统分别起一个消费组的名字，比如说：stock\_consumer\_group, marketing\_consumer\_group, credie\_consumer\_group, wms\_consumer\_group。

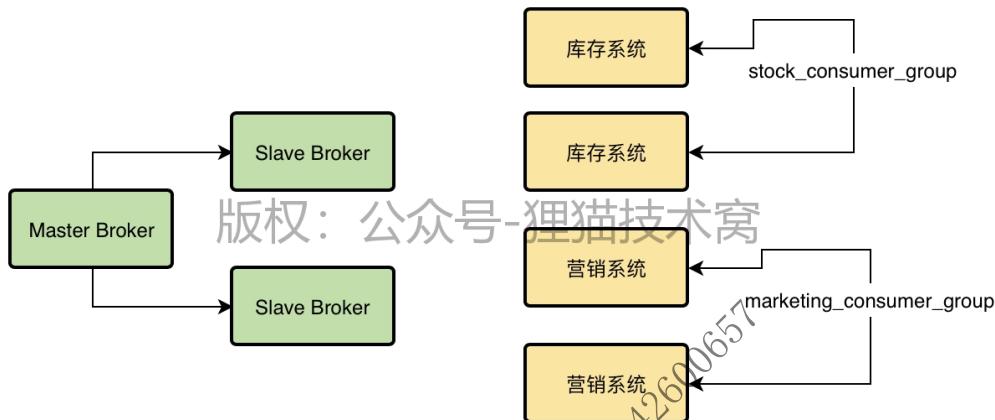
设置消费组的方式是在代码里进行的，类似下面这样：

```
1 DefaultMQPushConsumer consumer =
2     new DefaultMQPushConsumer("stock_consumer_group");
```

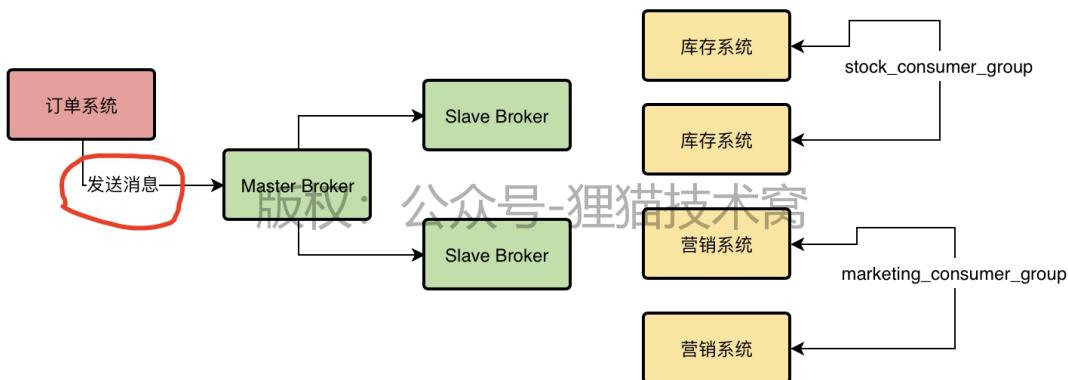
返回  
前进  
重新加载  
打印

然后比如库存系统部署了4台机器，每台机器上的消费者组的名字都是“stock\_consumer\_group”就同属于一个消费者组，以此类推，每个系统的几台机器都是属于各自的消费者组的。

我们看一下下面的图，里面我示意了两个系统，每个系统都有2台机器，每个系统都有一个自己的消费组。



然后给大家先解释一下不同消费者之间的关系，假设库存系统和营销系统作为两个消费者组，都订阅了“TopicOrderPaySuccess”这个订单支付成功消息的Topic，那么此时假设订单系统作为生产者发送了一条消息到这个Topic，如下图所示。



那么此时这条消息是怎么被消费的呢？

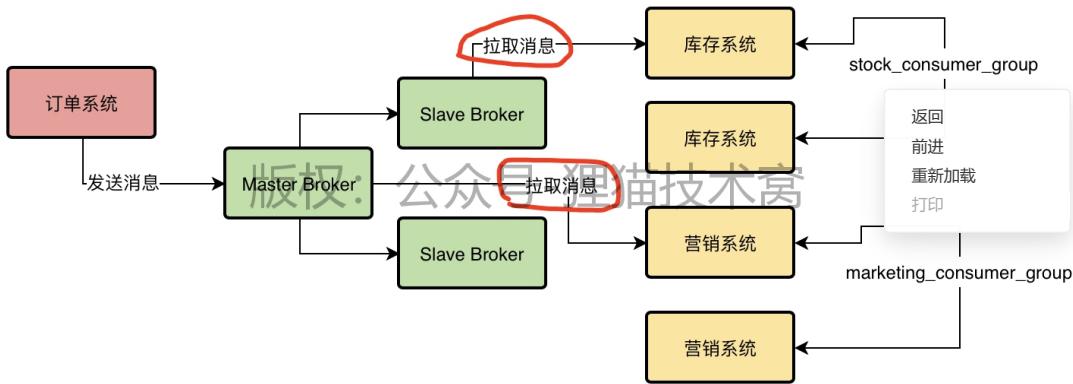
其实正常情况下来说，这条消息进入Broker之后，库存系统和营销系统作为两个消费组，每个组都会拉取到这条消息。

也就是说这个订单支付成功的消息，库存系统会获取到一条，营销系统也会获取到一条，他们俩都会获取到这条消息。

但是下一个问题来了，库存系统这个消费组里，他有两台机器，是两台机器都获取到这条消息？还是说只有一台机器会获取到这条消息？

答案是，正常情况下来说，库存系统的两台机器中只有一台机器会获取到这条消息，营销系统也是同理。

我们看下面的图，示意了对于一条订单支付成功的消息，库存系统的一台机器获取到了，营销系统的一台机器也获取到了。当然为了画图方便，图里是让营销系统从Master Broker拉取的，库存系统从Slave Broker拉取的。



这就是在消费的时候我们要给大家介绍的第一个知识点，不同的系统应该设置不同的消费组，如果不同的消费组订阅了同一个Topic，对Topic里的一条消息，每个消费组都会获取到这条消息。

## 2、集群模式消费 vs 广播模式消费

接着我们给大家介绍下一个概念，就是对于一个消费组而言，他获取到一条消息之后，如果消费组内部有多台机器，到底是只有一台机器可以获取到这个消息，还是每台机器都可以获取到这个消息？

这个就是集群模式和广播模式的区别。

默认情况下我们都是集群模式，也就是说，一个消费组获取到一条消息，只会交给组内的一台机器去处理，不是每台机器都可以获取到这条消息的。

但是我们可以通过如下设置来改变为广播模式：

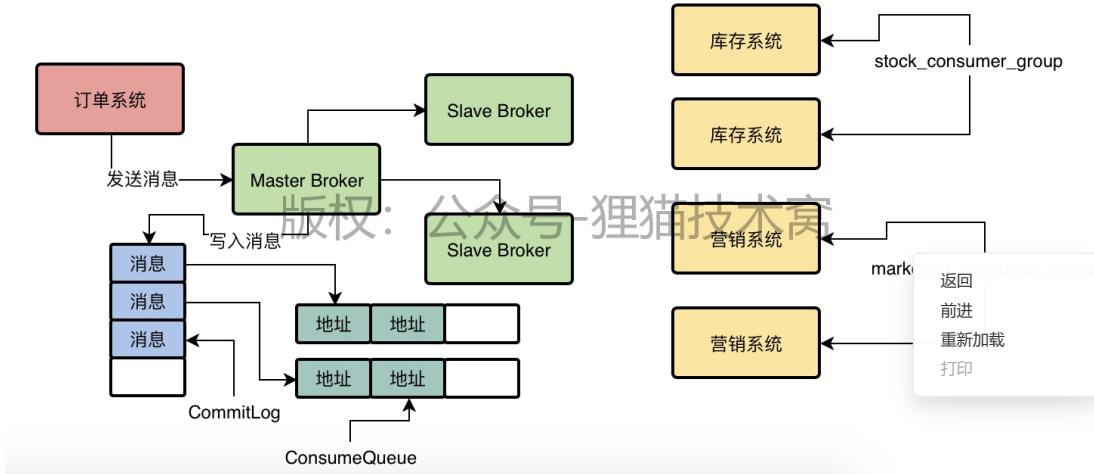
```
consumer.setMessageModel(MessageModel.BROADCASTING);
```

如果修改为广播模式，那么对于消费组获取到的一条消息，组内每台机器都可以获取到这条消息。但是相对而言广播模式其实用的很少，常见基本上都是使用集群模式来进行消费的。

## 3、重温MessageQueue、CommitLog、ConsumeQueue之间的关系

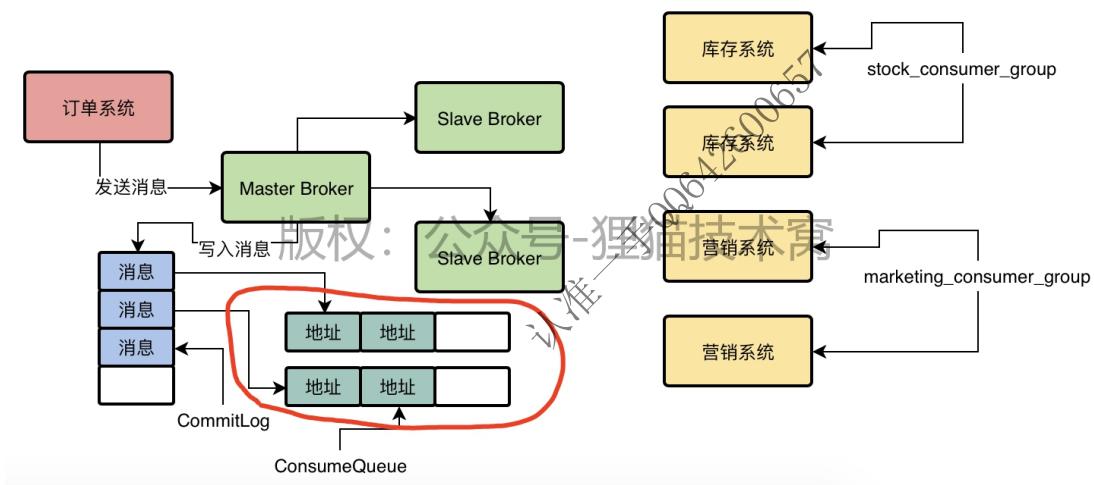
接着我们来研究一下MessageQueue与消费者的关系，通过之前的学习我们都已经知道了，一个Topic在创建的时候我们是要设置他有多少个MessageQueue的，而且我们也知道了，在Broker上MessageQueue是如何跟ConsumeQueue对应起来的。

我们先在图中展示出来这些概念，在Broker上我们会看到CommitLog文件，还有对应的多个ConsumeQueue文件。

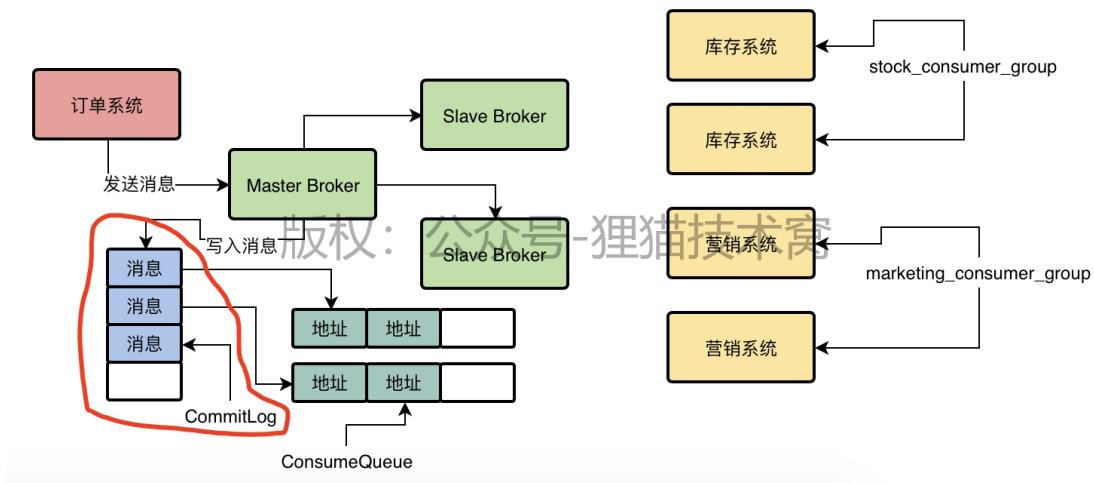


根据之前学习到的知识，我们大致可以如此理解，Topic中的多个MessageQueue会分散在多个Broker上，在每个Broker机器上，一个MessageQueue就对应了一个ConsumeQueue，当然在物理磁盘上其实是对应了多个ConsumeQueue文件的，但是我们大致也理解为一一对应关系。

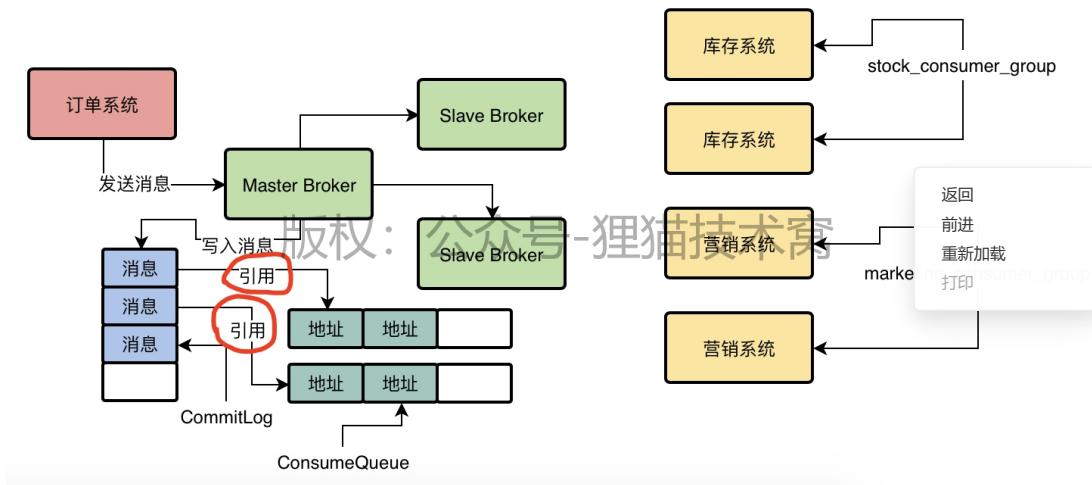
我们看下图中，我圈出来了ConsumeQueue，就代表了一个Topic的多个MessageQueue在物理磁盘上分别对应一个ConsumeQueue的意思。



但是对于一个Broker机器而言，存储在他上面的所有Topic以及MessageQueue的消息数据都是写入一个统一的CommitLog的，我们看下面的图，我圈出来了CommitLog，代表的是Broker上所有消息都是往里面写的。



然后对于Topic的各个MessageQueue而言，就是通过各个ConsumeQueue文件来存储属于MessageQueue的消息在CommitLog文件中的物理地址，就是一个offset偏移量，我在下面的图中标识出来了这个地址应用的关系。



#### 4、MessageQueue与消费者的关系

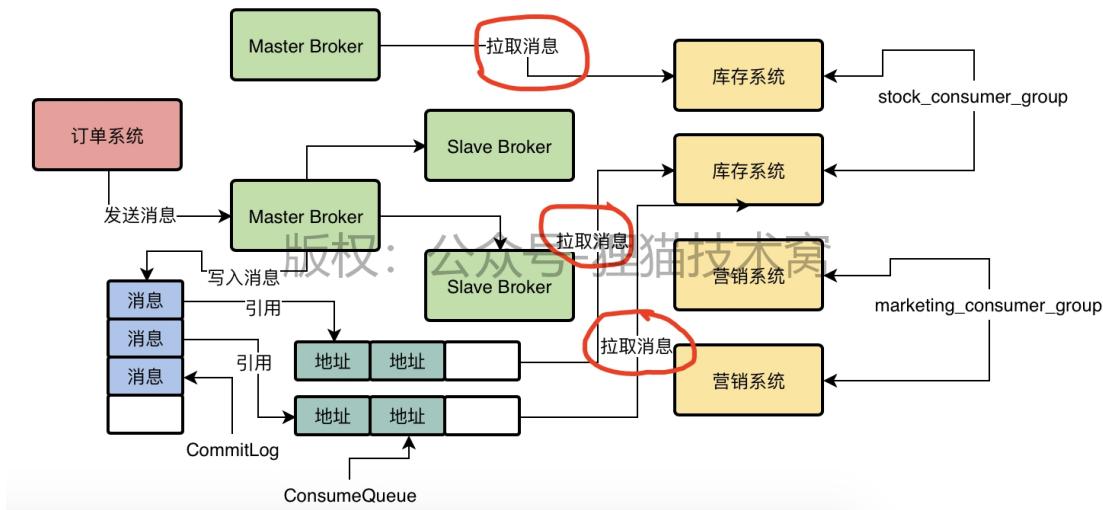
接着我们来想一个问题，对于一个Topic上的多个MessageQueue，是如何由一个消费组中的多台机器来进行消费的呢？

其实这里的源码实现细节是较为复杂的，但是我们可以简单的大致理解为，他会均匀的将MessageQueue分配给消费组的多台机器来消费。

举个例子，假设我们的“TopicOrderPaySuccess”有4个MessageQueue，这4个MessageQueue分布在两个Master Broker上，每个Master Broker上有2个MessageQueue。

然后库存系统作为一个消费组里有两台机器，那么正常情况下，当然最好的就是让这两台机器每个都负责2个MessageQueue的消费了，比如库存系统的机器01从Master Broker01上消费2个MessageQueue，然后库存系统的机器02从Master Broker02上消费2个MessageQueue，这样不就把消费的负载均摊到两台Master Broker上去了？

我们在下面的图里画出了这个示意。



所以你大致可以认为一个Topic的多个MessageQueue会均匀分摊给消费组内的多个机器去消费，这里的一个原则就是，一个MessageQueue只能被一个消费机器去处理，但是台消费者机器可以负责多个MessageQueue的消息处理。

#### 5、Push模式 vs Pull模式

现在我们已经知道了一个消费组内的多台机器是分别负责一部分MessageQueue的消费的，那么既然如此，每台机器都必须去连接到对应的Broker，尝试消费里面的MessageQueue对应的消息了。

此时就要涉及到两种消费模式了，之前我们也提到过，一个是Push，一个是Pull。实际上，这两个消费模式本质是一样的，都是消费者机器主动发送请求到Broker机器去拉取一批消息下来。

Push消费模式本质底层也是基于这种消费者主动拉取的模式来实现的，只不过他的名字叫做Push而已，意思是Broker会尽可能实时的把新消息交给消费者机器来进行处理，他的消息时效性会更好。

一般我们使用RocketMQ的时候，消费模式通常都是基于他的Push模式来做的，因为Pull模式的代码实现相对复杂和繁琐，而且Push模式底层本身就是基于消息拉取的方式来做的，只不过时效性更好而已。

Push模式的实现思路我们这里简单说一下：当消费者发送请求到Broker去拉取消息的时候，如果Broker处理完了，那么就会立马返回一批消息到消费机器去处理，处理完之后会接着立刻发送请求到Broker机器去拉取下一批消息。

所以消费机器在Push模式下会处理完一批消息，立马发起请求拉取下一批消息，消息处理的时效性非常好，看起来就跟Broker一直不停的推送消息到消费机器一样。

另外Push模式下有一个请求挂起和长轮询的机制，也要给大家简单介绍一下。

当你的请求发送到Broker，结果他发现没有新的消息给你处理的时候，就会让请求线程挂起，默认是挂起15秒，然后这个期间他会有后台线程每隔一会儿就去检查一下是否有的新的消息给你，另外如果在这个挂起过程中，如果有新的消息到达了会主动唤醒挂起的线程，然后把消息返回给你。

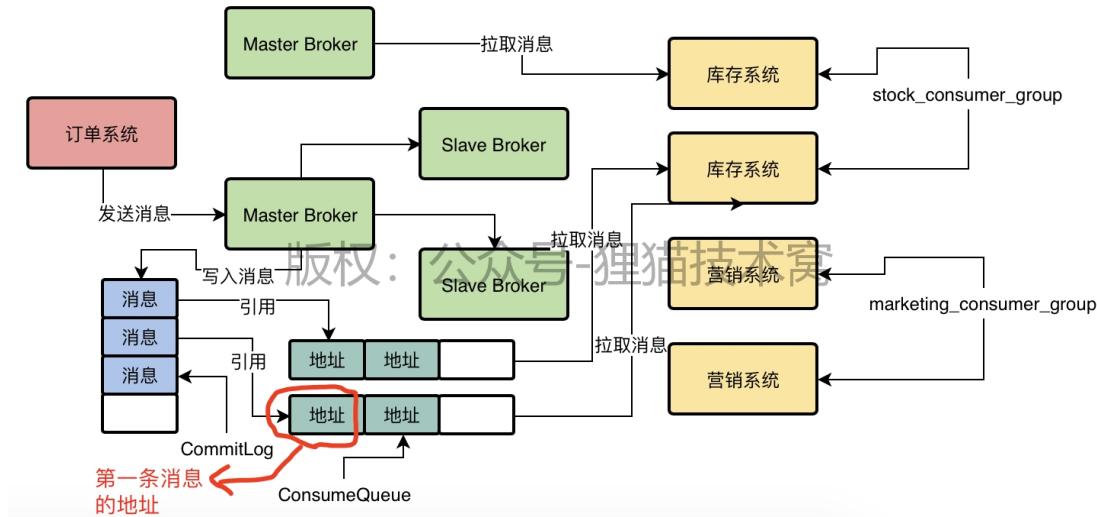
当然其实消费者进行消息拉取的底层源码是非常复杂的，涉及到大量的细节，但是他的核心思路大致就是如此，我们只要知道，其实哪怕是用常见的Push模式消费，本质也是消费者不停的发送请求到broker去拉取一批一批的消息就行了。

## 6、Broker是如何将消息读取出来返回给消费机器的？

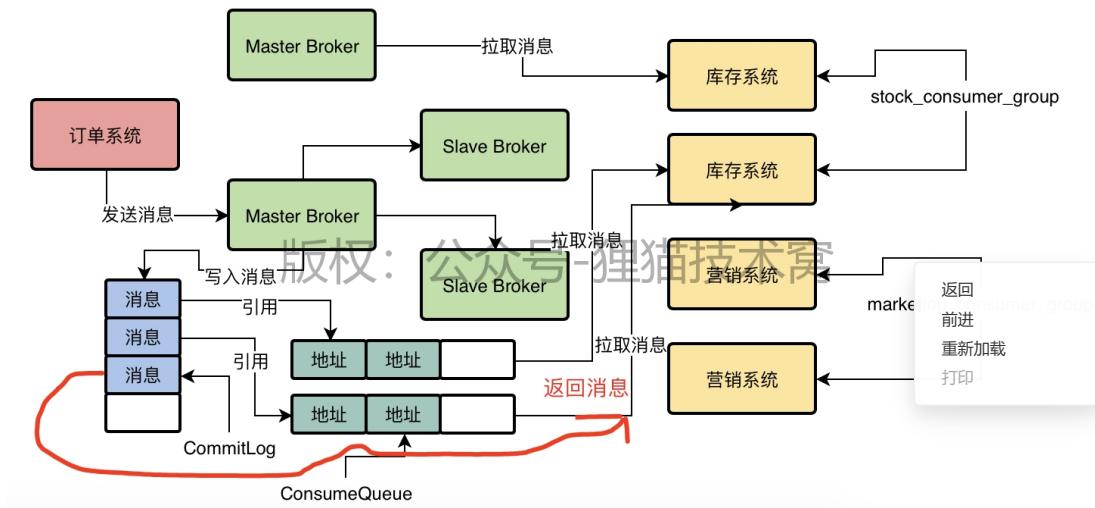
接着我们思考一个小问题，Broker在收到消费机器的拉取请求之后，是如何将消息读取出来返回给消费机器的？其实这里要涉及到两个概念，分别是ConsumeQueue和CommitLog。

假设一个消费者机器发送了拉取请求到Broker了，他说我这次要拉取MessageQueue0中的消息，然后我之前都没拉取过消息，所以就从这个MessageQueue0中的第一条消息开始拉取好了。

于是，Broker就会找到MessageQueue0对应的ConsumeQueue0，从里面找到第一条消息的offset，如下图所示。



接着Broker就需要根据ConsumeQueue中找到的第一条消息的地址，去CommitLog中根据这个offset地址去读取出这条消息的数据，然后把这条消息的数据返回给消费者机器，如下图所示。



所以其实消费消息的时候，本质就是根据你要消费的MessageQueue以及开始消费的位置，去找到对应的ConsumeQueue读取里面对应位置的消息在CommitLog中的物理offset偏移量，然后到CommitLog中根据offset读取消息数据，返回给消费者机器。

## 7、消费者机器如何处理消息、进行ACK以及提交消费进度？

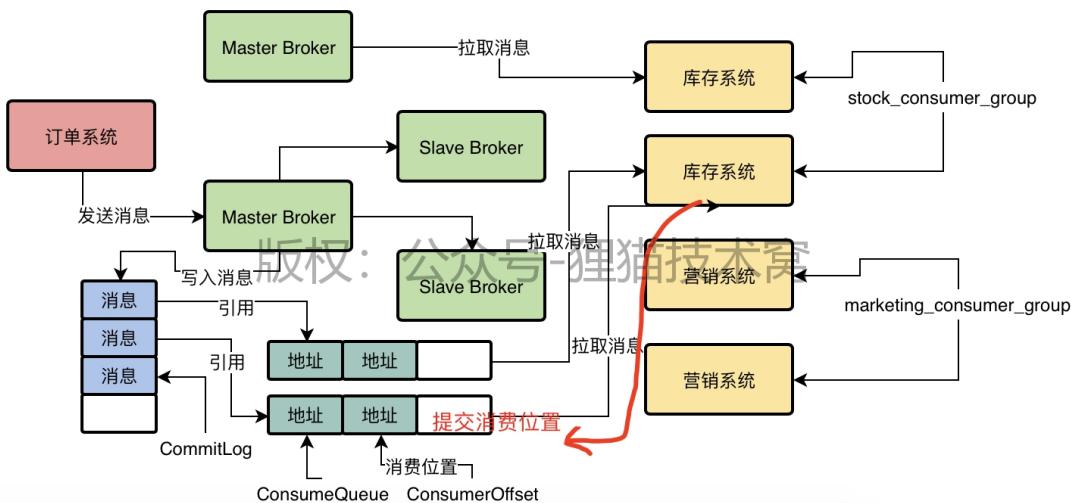
接着消费者机器拉取到一批消息之后，就会将这批消息回调我们注册的一个函数，如下这样子：

```

1 consumer.registerMessageListener(new MessageListenerConcurrently() {
2     @Override
3     public ConsumeConcurrentlyStatus consumeMessage(
4         List<MessageExt> msgs, ConsumeConcurrentlyContext context) {
5         // 处理消息
6         // 标记该消息已经被成功消费
7         return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
8     }
9 });

```

当我们处理完这批消息之后，消费者机器就会提交我们目前的一个消费进度到Broker上去，然后Broker就会存储我们的消费进度，比如我们现在对ConsumeQueue0的消费进度假设就是在offset=1的位置，那么他会记录下来一个ConsumerOffset的东西去标记我们的消费进度，如下图。



那么下次这个消费组只要再次拉取这个ConsumeQueue的消息，就可以从Broker记录的消费位置开始继续拉取，不用重头开始拉取了。

## 8、如果消费组中出现机器宕机或者扩容加机器，会怎么处理？

最后我们来看一下，如果消费组中出现机器宕机或者扩容加机器的情况，他会怎么处理？

这个时候其实会进入一个`rebalance`的环节，也就是说重新给各个消费机器分配他们要处理的`MessageQueue`。

给大家举个例子，比如现在机器01负责`MessageQueue0`和`Message1`，机器02负责`MessageQueue2`和`MessageQueue3`，现在机器02宕机了，那么机器01就会接管机器02之前负责的`MessageQueue2`和`MessageQueue3`。

或者如果此时消费组加入了一台机器03，此时就可以把机器02之前负责的`MessageQueue3`转移给机器03，然后机器01就仅仅负责一个`MessageQueue2`的消费了，这就是负载重平衡的概念。

## 9、消费源码流程提示

返回  
前进  
重新加载  
打印

我们最后就对消费这块的源码流程做一点提示，首先，拉取消息的源码入口是在`DefaultMQPushConsumerImpl`类的`pullMessage()`方法中的，这个里面涉及到了拉取请求、消息流量控制、通过`PullAPIWrapper`与服务端进行网络交互、服务端根据`ConsumeQueue`文件拉取消息，等一系列的事情，大家如果要分析`Consumer`端拉取消息的流程，可以从这个方法入口去看看。

到此为止，其实我们的MQ专栏整体就完成了，下次最后一篇文章，我会给大家做一个总结，同时告诉大家如何深度的炼化和理解本专栏的内容，以及把里面的内容运用到面试中去。

End

专栏版权归公众号**狸猫技术窝**所有

未经许可不得传播，如有侵权将追究法律责任

狸猫技术窝精品专栏及课程推荐：

- [《从零开始带你成为JVM实战高手》](#)
- [《21天互联网Java进阶面试训练营》（分布式篇）](#)
- [《互联网Java工程师面试突击》（第1季）](#)
- [《互联网Java工程师面试突击》（第3季）](#)

**重要说明：**

- 如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑
- 如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

具体加群方式，请参见目录菜单下的文档：《付费用户如何加群》（**购买后可见**）

返回  
前进  
重新加载  
打印

Copyright © 2015-2020 深圳小鹅网络技术有限公司 All Rights Reserved. [粤ICP备15020529号](#)

 小鹅通提供技术支持