

20 | RocketMQ Producer源码分析：消息生产的实现过程

李玥 · 消息队列高手课



你好，我是李玥。


对于消息队列来说，它最核心的功能就是收发消息。也就是消息生产和消费这两个流程。我们在之前的课程中提到了消息队列一些常见问题，比如，“如何保证消息不会丢失？”“为什么会收到重复消息？”“消费时为什么要先执行消费业务逻辑再确认消费？”，针对这些问题，我讲过它们的实现原理，这些最终落地到代码上，都包含在这一收一发两个流程中。

在接下来的两节课中，我会带你一起通过分析源码的方式，详细学习一下这两个流程到底是如何实现的。你在日常使用消息队列的时候，遇到的大部分问题，更多的是跟 Producer 和 Consumer，也就是消息队列的客户端，关联更紧密。搞清楚客户端的实现原理和代码中的细节，将对你日后使用消息队列时进行问题排查有非常大的帮助。所以，我们这两节课的重点，也将放在分析客户端的源代码上。

秉着先易后难的原则，我们选择代码风格比较简明易懂的 RocketMQ 作为分析对象。一起分析 RocketMQ 的 Producer 的源代码，学习消息生产的实现过程。

在分析源代码的过程中，我们的首要目的就是搞清楚功能的实现原理，另外，最好能有敏锐的嗅觉，善于发现代码中优秀的设计和巧妙构思，学习、总结并记住这些方法。在日常开发中，再遇到类似场景，你就可以直接拿来使用。

我们使用当前最新的 release 版本 release-4.5.1 进行分析，使用 Git 在 GitHub 上直接下载源码到本地：

 复制代码

```
1 git clone git@github.com:apache/rocketmq.git
2 cd rocketmq
3 git checkout release-4.5.1
```

客户端是一个单独的 Module，在 rocketmq/client 目录中。

从单元测试看 Producer API 的使用

在专栏之前的课程《[09 | 学习开源代码该如何入手？](#)》中我和你讲过，不建议你从 main() 方法入手去分析源码，而是带着问题去分析。我们本节课的问题是非常清晰的，就是要搞清楚 Producer 是如何发消息的。带着这个问题，接下来我们该如何分析源码呢？

我的建议是，先看一下单元测试用例。因为，一般单元测试中，每一个用例就是测试代码中的一个局部或者说是一个小流程。那对于一些比较完善的开源软件，它们的单元测试覆盖率都非常高，很容易找到我们关心的那个流程所对应的测试用例。我们的源码分析，就可以从这些测试用例入手，一步一步跟踪其方法调用链路，理清实现过程。

首先我们先分析一下 RocketMQ 客户端的单元测试，看看 Producer 提供哪些 API，更重要的是了解这些 API 应该如何使用。

Producer 的所有测试用例都在同一个测试

类"org.apache.rocketmq.client.producer.DefaultMQProducerTest"中，看一下这个测试类中的所有单元测试方法，大致可以了解到 Producer 的主要功能。

这个测试类的主要测试方法如下：

```
init
terminate
testSendMessage_ZeroMessage
testSendMessage_NoNameSrv
testSendMessage_NoRoute
testSendMessageSync_Success
testSendMessageSync_WithBodyCompressed
testSendMessageAsync_Success
testSendMessageAsync
testSendMessageAsync_BodyCompressed
testSendMessageSync_SuccessWithHook
```

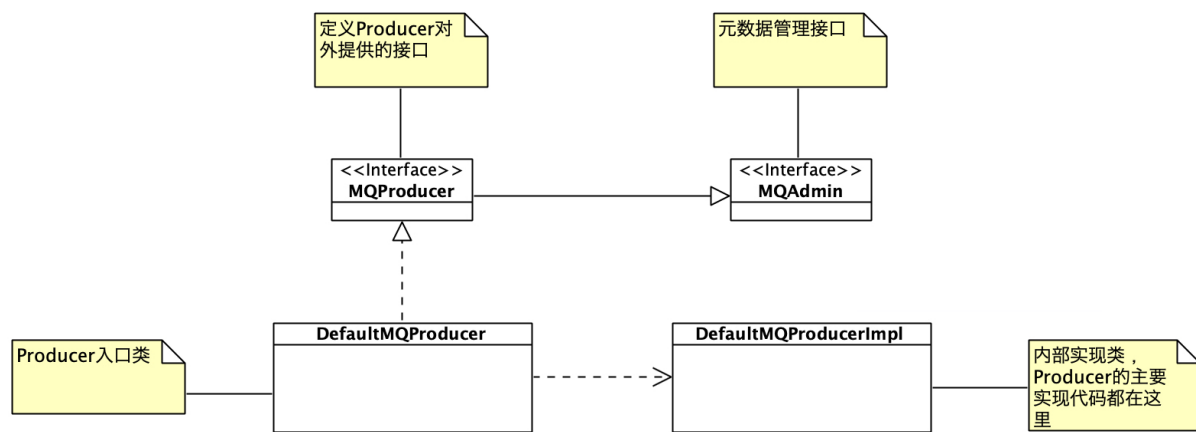
其中 `init` 和 `terminate` 是测试开始初始化和测试结束销毁时需要执行的代码，其他以 `testSendMessage` 开头的方法都是在各种情况和各种场景下发送消息的测试用例，通过这些用例的名字，你可以大致看出测试的功能。

比如，`testSendMessageSync` 和 `testSendMessageAsync` 分别是测试同步发送和异步发送的用例，`testSendMessageSync_WithBodyCompressed` 是压缩消息发送的测试用例，等等。

像 RocketMQ 这种开源项目，前期花费大量时间去编写测试用例，看似浪费时间，实际上会节省非常多后期联调测试、集成测试、以及上线后出现问题解决问题的时间，并且能够有效降低线上故障的概率，总体来说是非常划算的。强烈建议你在日常进行开发的过程中，也多写一些测试用例，尽量把单元测试的覆盖率做到 50% 以上。

RocketMQ 的 Producer 入口类

为“`org.apache.rocketmq.client.producer.DefaultMQProducer`”，大致浏览一下代码和类的继承关系，我整理出 Producer 相关的几个核心类和接口如下：



这里面 RocketMQ 使用了一个设计模式：门面模式（Facade Pattern）。

门面模式主要的作用是给客户端提供了一个可以访问系统的接口，隐藏系统内部的复杂性。

接口 MQProducer 就是这个模式中的门面，客户端只要使用这个接口就可以访问 Producer 实现消息发送的相关功能，从使用层面上来说，不必再与其他复杂的实现类打交道了。

类 DefaultMQProducer 实现了接口 MQProducer，它里面的方法实现大多没有任何的业务逻辑，只是封装了对其他实现类的方法调用，也可以理解为是门面的一部分。Producer 的大部分业务逻辑的实现都在类 DefaultMQProducerImpl 中，这个类我们会在后面重点分析其实现。

有的时候，我们的实现分散在很多的内部类中，不方便用接口来对外提供服务，你就可以仿照 RocketMQ 的这种方式，使用门面模式来隐藏内部实现，对外提供服务。


接口 MQAdmin 定义了一些元数据管理的方法，在消息发送过程中会用到。

启动过程

通过单元测试中的代码可以看到，在 init() 和 terminate() 这两个测试方法中，分别执行了 Producer 的 start 和 shutdown 方法，说明在 RocketMQ 中，Producer 是一个有状态的服

务，在发送消息之前需要先启动 Producer。这个启动过程，实际上就是为了发消息做的准备工作，所以，在分析发消息流程之前，我们需要先理清 Producer 中维护了哪些状态，在启动过程中，Producer 都做了哪些初始化的工作。有了这个基础才能分析其发消息的实现流程。


首先从测试用例的方法 init() 入手：

 复制代码

```
1  @Before
2  public void init() throws Exception {
3      String producerGroupTemp = producerGroupPrefix + System.currentTimeMillis()
4      producer = new DefaultMQProducer(producerGroupTemp);
5      producer.setNamesrvAddr("127.0.0.1:9876");
6      producer.setCompressMsgBodyOverHowmuch(16);
7
8      //省略构造测试消息的代码
9
10     producer.start();
11
12     //省略用于测试构造mock的代码
13 }
```

这段初始化代码的逻辑非常简单，就是创建了一个 DefaultMQProducer 的实例，为它初始化一些参数，然后调用 start 方法启动它。接下来我们跟进 start 方法的实现，继续分析其初始化过程。

DefaultMQProducer#start() 方法中直接调用了 DefaultMQProducerImpl#start() 方法，我们直接来看这个方法的代码：

 复制代码

```
1  public void start(final boolean startFactory) throws MQClientException {
2      switch (this.serviceState) {
3          case CREATE_JUST:
4              this.serviceState = ServiceState.START_FAILED;
5
6              // 省略参数检查和异常情况处理的代码
7
8              // 获取MQClientInstance的实例mQClientFactory，没有则自动创建新的实例
9              this.mQClientFactory = MQClientManager.getInstance().getAndCreateMQCl
10             // 在mQClientFactory中注册自己
```



```
11         boolean registerOK = mQClientFactory.registerProducer(this.defaultMQP
12         // 省略异常处理代码
13
14         // 启动mQClientFactory
15         if (startFactory) {
16             mQClientFactory.start();
17         }
18         this.serviceState = ServiceState.RUNNING;
19         break;
20     case RUNNING:
21     case START_FAILED:
22     case SHUTDOWN_ALREADY:
23         // 省略异常处理代码
24     default:
25         break;
26 }
27 // 给所有Broker发送心跳
28 this.mQClientFactory.sendHeartbeatToAllBrokerWithLock();
29 }
```

这里面，RocketMQ 使用一个成员变量 `serviceState` 来记录和管理自身的状态，这实际上是状态模式 (State Pattern) 这种设计模式的变种实现。

状态模式允许一个对象在其内部状态改变时改变它的行为，对象看起来就像是改变了它的类。

与标准的状态模式不同的是，它没有使用状态子类，而是使用分支流程 (switch-case) 来实现不同状态下的不同行为，在管理比较简单的状态时，使用这种设计会让代码更加简洁。这种模式非常广泛地用于管理有状态的类，推荐你在日常开发中使用。

在设计状态的时候，有两个要点是需要注意的，第一是，不仅要设计正常的状态，还要设计中间状态和异常状态，否则，一旦系统出现异常，你的状态就不准确了，你也就很难处理这种异常状态。比如在这段代码中，`RUNNING` 和 `SHUTDOWN_ALREADY` 是正常状态，`CREATE_JUST` 是一个中间状态，`START_FAILED` 是一个异常状态。

第二个要点是，将这些状态之间的转换路径考虑清楚，并在进行状态转换的时候，检查上一个状态是否能转换到下一个状态。比如，在这里，只有处于 `CREATE_JUST` 状态才能转换为

RUNNING 状态，这样就可以确保这个服务是一次性的，只能启动一次。从而避免了多次启动服务而导致的各种问题。

接下来看一下启动过程的实现：

1. 通过一个单例模式（Singleton Pattern）的 MQClientManager 获取 MQClientInstance 的实例 mQClientFactory，没有则自动创建新的实例；
2. 在 mQClientFactory 中注册自己；
3. 启动 mQClientFactory；
4. 给所有 Broker 发送心跳。

这里面又使用了一个最简单的设计模式：单例模式。我们在这儿给出单例模式的定义，不再详细说明了，不会的同学需要自我反省一下，然后赶紧去复习设计模式基础去。

单例模式涉及一个单一的类，该类负责创建自己的对象，同时确保只有单个对象被创建。这个类提供了一种访问其唯一的对象的方式，可以直接访问，不需要实例化该类的对象。

其中实例 mQClientFactory 对应的类 MQClientInstance 是 RocketMQ 客户端中的顶层类，大多数情况下，可以简单地理解为每个客户端对应类 MQClientInstance 的一个实例。这个实例维护着客户端的大部分状态信息，以及所有的 Producer、Consumer 和各种服务的实例，想要学习客户端整体结构的同学可以从分析这个类入手，逐步细化分析下去。

我们进一步分析一下 MQClientInstance#start() 中的代码：

```
1 // 启动请求响应通道
2 this.mQClientAPIImpl.start();
3 // 启动各种定时任务
4 this.startScheduledTask();
5 // 启动拉消息服务
6 this.pullMessageService.start();
7 // 启动Rebalance服务
8 this.rebalanceService.start();
9 // 启动Producer服务
10 this.defaultMQProducer.getDefaultMQProducerImpl().start(false);
```

这一部分代码的注释比较清楚，流程是这样的：

1. 启动实例 mQClientAPIImpl，其中 mQClientAPIImpl 是类 MQClientAPIImpl 的实例，封装了客户端与 Broker 通信的方法；
2. 启动各种定时任务，包括与 Broker 之间的定时心跳，定时与 NameServer 同步数据等任务；
3. 启动拉取消息服务；
4. 启动 Rebalance 服务；
5. 启动默认的 Producer 服务。

以上是 Producer 的启动流程。这里面有几个重要的类，你需要清楚它们的各自的职责。后续你在使用 RocketMQ 时，如果遇到问题需要调试代码，了解这几个重要类的职责会对你有非常大的帮助。

1. DefaultMQProducerImpl：Producer 的内部实现类，大部分 Producer 的业务逻辑，也就是发消息的逻辑，都在这个类中。
2. MQClientInstance：这个类中封装了客户端一些通用的业务逻辑，无论是 Producer 还是 Consumer，最终需要与服务端交互时，都需要调用这个类中的方法；
3. MQClientAPIImpl：这个类中封装了客户端服务端的 RPC，对调用者隐藏了真正网络通信部分的具体实现；

4. NettyRemotingClient: RocketMQ 各进程之间网络通信的底层实现类。

消息发送过程

接下来我们一起分析 Producer 发送消息的流程。


在 Producer 的接口 MQProducer 中，定义了 19 个不同参数的发消息的方法，按照发送方式不同可以分成三类：

单向发送（Oneway）：发送消息后立即返回，不处理响应，不关心是否发送成功；

同步发送（Sync）：发送消息后等待响应；

异步发送（Async）：发送消息后立即返回，在提供的回调方法中处理响应。

这三类发送实现基本上是相同的，异步发送稍微有一点儿区别，我们看一下异步发送的实现方法"DefaultMQProducerImpl#send()"（对应源码中的 1132 行）：

 复制代码

```
1  @Deprecated
2  public void send(final Message msg, final MessageQueueSelector selector, final Ob
3      throws MQClientException, RemotingException, InterruptedException {
4      final long beginStartTime = System.currentTimeMillis();
5      ExecutorService executor = this.getAsyncSenderExecutor();
6      try {
7          executor.submit(new Runnable() {
8              @Override
9              public void run() {
10                 long costTime = System.currentTimeMillis() - beginStartTime;
11                 if (timeout > costTime) {
12                     try {
13                         try {
14                             sendSelectImpl(msg, selector, arg, CommunicationMode.
15                                 timeout - costTime);
16                         } catch (MQBrokerException e) {
17                             throw new MQClientException("unknownn exception", e);
18                         }
19                     } catch (Exception e) {
20                         sendCallback.onException(e);
21                     }
22                 } else {
23                     sendCallback.onException(new RemotingTooMuchRequestException(
```


```

24         }
25     }
26
27     });
28     } catch (RejectedExecutionException e) {
29         throw new MQClientException("executor rejected ", e);
30     }
31 }

```

我们可以看到，RocketMQ 使用了一个 ExecutorService 来实现异步发送：使用 asyncSenderExecutor 的线程池，异步调用方法 sendSelectImpl()，继续发送消息的后续工作，当前线程把发送任务提交给 asyncSenderExecutor 就可以返回了。单向发送和同步发送的实现则是直接在当前线程中调用方法 sendSelectImpl()。

我们来继续看方法 sendSelectImpl() 的实现：

 复制代码

```

1  // 省略部分代码
2  MessageQueue mq = null;
3
4  // 选择将消息发送到哪个队列 (Queue) 中
5  try {
6      List<MessageQueue> messageQueueList =
7          mqClientFactory.getMQAdminImpl().parsePublishMessageQueues(topicPublishIn
8      Message userMessage = MessageAccessor.cloneMessage(msg);
9      String userTopic = NamespaceUtil.withoutNamespace(userMessage.getTopic(), mQC
10     userMessage.setTopic(userTopic);
11
12     mq = mqClientFactory.getClientConfig().queueWithNamespace(selector.select(mes
13 } catch (Throwable e) {
14     throw new MQClientException("select message queue threw exception.", e);
15 }
16
17 // 省略部分代码
18
19 // 发送消息
20 if (mq != null) {
21     return this.sendKernelImpl(msg, mq, communicationMode, sendCallback, null, ti
22 } else {
23     throw new MQClientException("select message queue return null.", null);
24 }
25 // 省略部分代码

```

方法 `sendSelectImpl()` 中主要的功能就是选定要发送的队列，然后调用方法 `sendKernellImpl()` 发送消息。

选择哪个队列发送由 `MessageQueueSelector#select` 方法决定。在这里 RocketMQ 使用了策略模式（Strategy Pattern），来解决不同场景下需要使用不同的队列选择算法问题。

策略模式：定义一系列算法，将每一个算法封装起来，并让它们可以相互替换。策略模式让算法独立于使用它的客户而变化。

RocketMQ 提供了很多 `MessageQueueSelector` 的实现，例如随机选择策略，哈希选择策略和同机房选择策略等，如果需要，你也可以自己实现选择策略。之前我们的课程中提到过，如果要保证相同 key 消息的严格顺序，你需要使用哈希选择策略，或者提供一个自己实现的选择策略。

接下来我们再看一下方法 `sendKernellImpl()`。这个方法的代码非常多，大约有 200 行，但逻辑比较简单，主要功能就是构建发送消息的头 `RequestHeader` 和上下文 `SendMessageContext`，然后调用方法 `MQClientAPIImpl#sendMessage()`，将消息发送给队列所在的 Broker。

至此，消息被发送给远程调用的封装类 `MQClientAPIImpl`，完成后续序列化和网络传输等步骤。

可以看到，RocketMQ 的 Producer 整个发消息的流程，无论是同步发送还是异步发送，都统一到同一个流程中。包括异步发送消息的实现，实际上也是通过一个线程池，在异步线程执行的调用和同步发送相同的底层方法来实现的。

在底层方法的代码中，依靠方法的一个参数来区分同步还是异步发送。这样实现的好处是，整个流程是统一的，很多同步异步共同的逻辑，代码可以复用，并且代码结构清晰简单，便于维护。

使用同步发送的时候，当前线程会阻塞等待服务端的响应，直到收到响应或者超时方法才会返回，所以在业务代码调用同步发送的时候，只要返回成功，消息就一定发送成功了。异步发送

的时候，发送的逻辑都是在 Executor 的异步线程中执行的，所以不会阻塞当前线程，当服务端返回响应或者超时之后，Producer 会调用 Callback 方法来给业务代码返回结果。业务代码需要在 Callback 中来判断发送结果。这和我们在之前的课程《[🔗05 | 如何确保消息不会丢失？](#)》讲到的发送流程是完全一样的。

小结

这节课我带你分析了 RocketMQ 客户端消息生产的实现过程，包括 Producer 初始化和发送消息的主流程。Producer 中包含的几个核心的服务都是有状态的，在 Producer 启动时，在 MQClientInstance 这个类中来统一来启动。在发送消息的流程中，RocketMQ 分了三中发送方式：单向、同步和异步，这三种发送方式对应的发送流程基本是相同的，同步和异步发送是由已经封装好的 MQClientAPIImpl 类来分别实现的。

对于我们在分析代码中提到的几个重要的业务逻辑实现类，你最好能记住这几个类和它的功能，包括：DefaultMQProducerImpl 封装了大部分 Producer 的业务逻辑，MQClientInstance 封装了客户端一些通用的业务逻辑，MQClientAPIImpl 封装了客户端与服务端的 RPC，NettyRemotingClient 实现了底层网络通信。

我在课程中，只能带你把主干流程分析清楚，但是很多细节并没有涉及，课后请你一定要按照流程把源代码仔细看一遍，仔细消化一下没有提及到的分支流程，将这两个流程绘制成详细的流程图或者时序图。

分析过程中提到的几个设计模式，是非常实用且常用的设计模式，希望你能充分理解并熟练运用。

思考题

你有没有注意到，在源码中，异步发送消息方法 DefaultMQProducerImpl#send()(1132 行) 被开发者加了 @Deprecated（弃用）注解，显然开发者也意识到了这种异步的实现存在一些问题，需要改进。请你结合我们专栏文章《[🔗10 | 如何使用异步设计提升系统性能？](#)》中讲到的异步设计方法想一想，应该如何改进这个异步发送的流程？欢迎在留言区写下你的想法。

感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给你的朋友。

精选留言 (31)



. . o O

2019-10-28

请教老师一个问题,如果异步发送的话,就是把发送逻辑封装成任务放到线程池里去处理,那么是不是就没法保证消息的顺序性了呢?哪怕是通过key哈希到一个同一个队列,但是发送消息的任务执行先后顺序没法保证吧?

作者回复: 异步发送仍然可以保证严格顺序, 但需要注意几点:

1. 需要单线程异步发送;
2. 需要记录一个递增流水号, 保证每个发出的消息都有一个流水号, 如果某个流水号的消息发送出错, 需要重发这个流水号之后的所有消息。比如, 连续异步发送12345这5条消息, 假如已经异步发送了12345, 然后异步检查发送结果的时候发现3发送失败了, 需要从3开始重发。也就是按顺序重发345。
3. 消费逻辑需要幂等, 能接受2中的这种情况, 也就是说, 收到的消息有可能是: 12 (3丢了) 45 34 5。

共 2 条评论 >

👍 34



微微一笑

2019-09-11

老师好, 先祝您节日快乐!!! 您辛苦了~

有几个疑问需要老师解答一下:

①今天在看rocketMq源码过程中, 发现DefaultMQProducer有个属性defaultTopicQueueNums, 它是用来设置topic的ConsumeQueue的数量的吗? 我之前的理解是, consumeQueue的数量是创建topic的时候指定的, 跟producer没有关系, 那这个参数又有什么作用呢?

②在RocketMq的控制台上可以创建topic, 需要指定writeQueueNums, readQueueNums, perm, 这三个参数是有什么用呢? 这里为什么要区分写队列跟读队列呢? 不应该只有一个consumeQueue吗?

③用户请求-->异步处理--->用户收到响应结果。异步处理的作用是: 用更少的线程来接收更多的用户请求, 然后异步处理业务逻辑。老师, 异步处理完后, 如何将结果通知给原先的用户呢? 即使有回调接口, 我理解也是给用户发个短信之类的处理, 那结果怎么返回到定位到用

户，并返回之前请求的页面上呢？需要让之前的请求线程阻塞吗？那也无法达到【用更少的线程来接收更多的用户请求】的目的丫。

望老师能指点迷津~~~

作者回复: A1: 这个参数是控制客户端在生产消费的时候会访问同一个主题的队列数量，假设一个主题有100个队列，对于每一个客户端来说，它没必要100个队列都访问，只需要使用其中的几个队列就行了。

A2: writeQueueNums和readQueueNums是在服务端来控制每个客户端在生产和消费的时候，分别访问多少个队列。这两个参数是服务端参数，优先级是高于客户端控制的参数defaultTopicQueueNums的。perm是设置Topic读写等权限的参数，具体如何设置你需要去看一下文档。

A3: 如果局限于：“APP/浏览器 --[http协议]-->web 服务”这样的场景，受限于http协议，前端和web服务的交互一定是单向和同步的。一定要等待结果然后返回响应，但是，这种情况仍然可以使用异步的方法，这个我在“08答疑”中解释秒杀的时候其实已经给出了答案。很多同学不理解的原因是思维被web框架给限制住了。像spring web这种框架，它把处理web请求都给你封装好了，你只要写一个handler就行了，很方便。但是，这个handler只能是一个同步方法，它必须在返回值中给出响应结果，所以导致很多同学的思维转不过来这个弯儿。

你可以结合我们讲的异步网络IO内容想一下，http协议发一个请求到服务端，就是发了一些数据过来，服务端回响应也就是在这个连接上给它返回一些数据回去就可以了。至于什么时候往回发响应数据，哪个线程来发，有要求吗？并没有。只要在超时之前发响应就可以了。我们讲得如何实现异步网络IO的方法处理的不就是这种情况吗？

这个过程不是说一定要做成和web框架一样的同步处理。

共 3 条评论 >

👍 22



Imtoo

2019-09-10

这种异步方式几乎没有意义，底层的netty已经实现了异步，这里只是在选择消息队列等判断的过程加了异步，最终callback还是由netty线程来调用的

共 8 条评论 >

👍 20



z.l

2019-11-18

老师，异步发送为什么是弃用，还是没有看懂，感觉超时时间的计算没有错啊...

作者回复: 我理解主要的原因除了超时时间的计算不准确以外, 更重要的原因是这个异步方法还有改进的空间, 其实可以直接结合Netty做到不需要Executor。



👍 13



Peter

2019-11-04

课后作业, 请老师指正:

从方法的注释看, 说是因为异常处理和超时时间的语义不对。

异常处理这块我觉得应该是采用统一的异常处理, 而不应该是有的异常抛出, 而有的异常通过回调方法返回客户端。

再说超时时间的错误语义, 严格来说应该是不准确的超时时间, 因为在run方法里进行时间判断 (if (timeout > costTime)) 实际上已经是开始执行当前线程的时间, 而之前的排队时间没有算, 因此我改进的方法应该是这样:

```
CompletableFuture.runAsync(() -> {
    long costTime = System.currentTimeMillis() - beginStartTime;
    if (timeout > costTime) {
        try {
            sendDefaultImpl(msg, CommunicationMode.ASYNC, sendCallback, timeout - costTime);
        } catch (Exception e) {
            sendCallback.onException(e);
        }
    } else {
        sendCallback.onException(
            new RemotingTooMuchRequestException("DEFAULT ASYNC send call timeout"));
    }
}, executor);
```

作者回复: 给认真思考完成作业的同学点赞👍!

共 7 条评论 >

👍 7



每天晒白牙

2019-09-10

我总结的kafka生产消息的源码分析

https://mp.weixin.qq.com/s/-s34_y16HU6HR5HDsSD4bg



👍 6



leslie

2019-09-10

编程语言的话Python或Go可以么？极客时间里都有购买，就是忙着其它课程的学习，一直没顾的上编程语言的学习。

从开始一路跟到现在：算是少数一直在完全没有缺的课；前期一直遍边学习边针对开篇时的学习目标针对当下工作环境的Nosql DB和MQ使用率的低下的问题找解决思路 and 方案，课后笔记主要同样集中在思路以及针对思路的困惑查疑上，代码这块完全没顾上。虽然代码的思路看的懂，发现动手能力确实非常欠缺。一路学到现在梳理到现在整体方案大致定下来：以及早期的部分课程的结束；课程的主要方案自己估计在掌握思路的基础上去补强Coding能力。虽然DBA的Coding能力都比较烂，不过还是得边学边啃下来；逼自己一下总能勉强写出来，估计就是效率问题、、、MQ这块PY或GO哪种更合适，或者说都可以？

感谢老师一路的辛勤授业：授课之余尽力去帮助学生们解惑，让我们能一路走来一路成长；愿老师教师节快乐，谢谢老师的分享。

作者回复: 个人建议学习Java或者Go，这两种语言都有不错的生态系统，都可以用来构建大规模集群。

相对来说，Java的生态系统更强大，Go比较年轻，有很多Java不具备的语言特性。

Python本来只是一门脚本语言，特别适合开发机器学习程序而火起来了，如果你不是从事机器学习相关的研发，不太建议作为第一语言来学习。



👍 6



墙角儿的花

2019-09-11

老师 对于im服务器集群，客户端的socket均布在各个服务器，目标socket不在同一个服务器上时，服务器间需要转发消息，这个场景需要低延迟无需持久化，服务器间用redis的发布订阅，因其走内存较快，即使断电还可以走库。im服务器和入库服务间用其他mq解耦，因为这个环节需要持久化，所以选rocketmq或kafka，但kafka会延迟批量发布消息 所以选rocketmq，这两个环节的mq选型可行吗。

作者回复: 有一个问题你需要考虑，你是不是需要为每一个会话（比如，张三和李四之间开始聊天，成为一个会话）在MQ中凑创建一个Topic呢？这样会导致MQ集群中的Topic数量非常多。假设你的

系统注册用户数是n，理论上最多会需要 $n \times n$ 个Topic，这还没有计算用户拉的群。

对于海量的Topic数量，RocketMQ和Kafka都不是太好的选择。

共 3 条评论 >

👍 5



明日

2019-09-10

李老师节日快乐！

关于思考题看到了源码的注释说异常处理和超时时间有问题。

自己看的话一是异常这里抛未知的原因，不够明确。

二是这里用的线程池默认使用了虚拟机可用的线程，可能会对其他服务造成影响。

三是超时时间这把线程阻塞可能等待的时间也包括进去了不太合适。

感觉代码层次使用老师说过的completablefuture处理更优雅。另外底层使用了netty，应该直接用异步io就行了吧。



👍 4



侧面

2020-01-14

有这篇这课就买的值了



👍 2



二少

2021-04-13

DefaultMQProducer是DefaultMQProducerImpl的门面，但二者的类名起得有点怪怪的感觉。类名有Impl后缀，一般都表示这个类是某个接口的实现类，但实际上却是门面和被包装类的关系。而且把门面类给个facade后缀不是更适当一些吗？大家怎么看。



👍 1



Heaven

2021-02-08

因为Netty本身就支持异步的写入消息,并注入Listener,这一步的发送,则是利用Nio的WorkGroup,这种情况下,显式的使用线程池异步的发送显得有点多余



👍 1



编程界的小学生

2020-07-01

RocketMQ同等的策略模式还有消费端的时候选择消费者与queue的对应策略：

AllocateMessageQueueStrategy接口下有如下几个实现类

AllocateMessageQueueAveragely

AllocateMachineRoomNearby

AllocateMessageQueueAveragelyByCircle

AllocateMessageQueueByConfig：这个策略真不知道有啥鸟用

AllocateMessageQueueByMachineRoom

AllocateMessageQueueConsistentHash

而且看这名字就知道是策略模式。直接以Strategy结尾。



1



我丢了一只小凳子

2020-02-28

跟踪源码发现，异步回调，最后还是在NettyRemotingAbstract中启动线程池做了

/**

* Execute callback in callback executor. If callback executor is null, run directly in current thread

*/

```
private void executeInvokeCallback(final ResponseFuture responseFuture) {
```

```
    boolean runInThisThread = false;
```

```
    ExecutorService executor = this.getCallbackExecutor();
```

```
    if (executor != null) {
```

```
        try {
```

```
            executor.submit(new Runnable() {
```

```
                @Override
```

```
                public void run() {
```

```
                    try {
```

```
                        responseFuture.executeInvokeCallback();
```

```
                    } catch (Throwable e) {
```

```
                        log.warn("execute callback in executor exception, and callback thro
```

```
w", e);
```

```
                    } finally {
```

```
                        responseFuture.release();
```

```
                    }
```

```
                }
```

```
            });
```

```
        } catch (Exception e) {
```

```
            runInThisThread = true;
```

```
        log.warn("execute callback in executor exception, maybe executor busy", e);
    }
} else {
    runInThisThread = true;
}

if (runInThisThread) {
    try {
        responseFuture.executeInvokeCallback();
    } catch (Throwable e) {
        log.warn("executeInvokeCallback Exception", e);
    } finally {
        responseFuture.release();
    }
}
}
```



1



Peter

2019-11-04

老师继续请教问题：

- 1.DefaultMQPullConsumer和DefaultMQPushConsumer有什么区别
- 2.为什么pullConsumer的启动和producer的启动在同一个start方法里（最终都在MQClientIn stance#start里）
- 3.rebalanceService服务是干嘛的

作者回复: PullConsumer：业务代码在需要的时候调用consumer.pullxxxx方法从consumer拉消息；
PushConsumer：当有消息的时候，consumer会自动调用messageListener(业务处理消息的代码)。

这两种方式主要是为了方便使用者进行线程控制，没有什么本质区别。



1



姑射仙人

2019-09-28

1. 异常处理问题：线程内部抛出的异常，比如MQBrokerException，客户端无法感知到，以为发送成功，会继续执行。

2. 超时时间的概念问题：这里似乎去掉了线程调度的时间，将剩下的时间给了netty，个人感觉也应该包含进去。对客户端而言，调度是自己的事，不应包含在网络超时时间里。

请老师指正。



谁都会变

2022-09-29 来自上海

消息生产者启动拉取消息这个感觉没什么用啊，它不是推送消息得吗？



fomy

2020-02-16

1、为什么ServiceState变量不设置成volatile呢？

2、消费者MessageQueue(readQueueNums)怎么和生产者MessageQueue(writeQueueNums)关联起来的呢？比如readQueueNums=19个，writeQueueNums=23个，它们是怎么关联的呢？

作者回复: A1: 因为它服务的类没有设计成线程安全的，所以也没必要用volatile关键字。

A2: writeQueueNums和readQueueNums是在服务端来控制每个客户端在生产和消费的时候，分别访问多少个队列。因为对主题来说，生产者的实例数和消费者的实例数是没有关系的，所以这两个参数是不关联的。



七楼

2020-01-02

mvc框架的 controller也算是门面模式的门面把？他也是提供一个可访问系统的借口 隐藏了系统内部的复杂性 对吗

共 1 条评论 >



z.l

2019-11-14

DefaultMQProducerImpl的start和shutdown方法没有加同步，serviceState也只是一个普通成员变量没加volatile，不会有线程安全问题吗？

作者回复: 这两个方法没有做到线程安全，但是这两个方法的实现内部，调用的方法都是线程安全的方法。

