20 | RocketMQ Producer源码分析：消息生产的实现过程

2019-09-10 李玥

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：

```
git clone git@github.com:apache/rocketmq.git
cd rocketmq
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

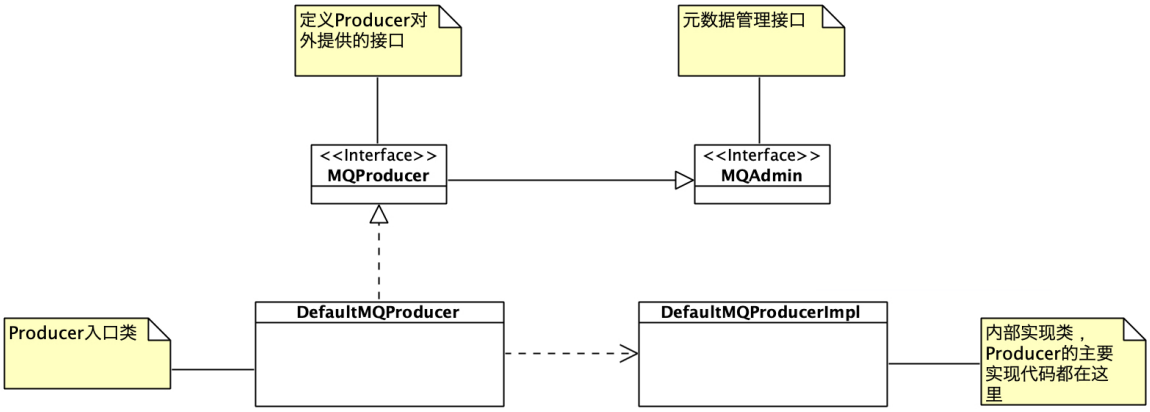
其中init和terminate是测试开始初始化和测试结束销毁时需要执行的代码，其他以

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。

RockectMQ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：

```
@Before
public void init() throws Exception {
    String producerGroupTemp = producerGroupPrefix + System.currentTimeMillis();
    producer = new DefaultMQProducer(producerGroupTemp);
    producer.setNamesrvAddr("127.0.0.1:9876");
    producer.setCompressMsgBodyOverHowmuch(16);

    //省略构造测试消息的代码

    producer.start();

    //省略用于测试构造mock的代码
}
```

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：

```
public void start(final boolean startFactory) throws MQClientException {
    switch (this.serviceState) {
        case CREATE_JUST:
            this.serviceState = ServiceState.START_FAILED;

            // 省略参数检查和异常情况处理的代码

            // 获取MQClientInstance的实例mQClientFactory，没有则自动创建新的实例
            this.mQClientFactory = MQClientManager.getInstance().getAndCreateMQClientInstance(this.defaultMQPr
            // 在mQClientFactory中注册自己
            boolean registerOK = mQClientFactory.registerProducer(this.defaultMQProducer.getProducerGroup(), this
            // 省略异常处理代码

            // 启动mQClientFactory
            if (startFactory) {
                mQClientFactory.start();
            }
            this.serviceState = ServiceState.RUNNING;
            break;
        case RUNNING:
        case START_FAILED:
        case SHUTDOWN_ALREADY:
            // 省略异常处理代码
        default:
            break;
    }
    // 给所有Broker发送心跳
    this.mQClientFactory.sendHeartbeatToAllBrokerWithLock();
}
```

这里面，RocketMQ使用一个成员变量serviceState来记录和管理自身的服务状态，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：

```
// 启动请求响应通道
this.mQClientAPIImpl.start();
// 启动各种定时任务
this.startScheduledTask();
// 启动拉消息服务
this.pullMessageService.start();
// 启动Rebalance服务
this.rebalanceService.start();
// 启动Producer服务
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）：

```
@Deprecated
public void send(final Message msg, final MessageQueueSelector selector, final Object arg, final SendCallback se
    throws MQClientException, RemotingException, InterruptedException {
    final long beginStartTime = System.currentTimeMillis();
    ExecutorService executor = this.getAsyncSenderExecutor();
    try {
        executor.submit(new Runnable() {
            @Override
            public void run() {
                long costTime = System.currentTimeMillis() - beginStartTime;
                if (timeout > costTime) {
                    try {
                        try {
                            sendSelectImpl(msg, selector, arg, CommunicationMode.ASYNC, sendCallback,
                                timeout - costTime);
                        } catch (MQBrokerException e) {
                            throw new MQClientException("unknownn exception", e);
                        }
                    } catch (Exception e) {
                        sendCallback.onException(e);
                    }
                } else {
                    sendCallback.onException(new RemotingTooMuchRequestException("call timeout"));
                }
            }

        });
    } catch (RejectedExecutionException e) {
        throw new MQClientException("exector rejected ", e);
    }
}
```

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：

```
// 省略部分代码
MessageQueue mq = null;

// 选择将消息发送到哪个队列（Queue）中
try {
    List<MessageQueue> messageQueueList =
        mQClientFactory.getMQAdminImpl().parsePublishMessageQueues(topicPublishInfo.getMessageQueueList()
    Message userMessage = MessageAccessor.cloneMessage(msg);
    String userTopic = NamespaceUtil.withoutNamespace(userMessage.getTopic(), mQClientFactory.getClientConf
    userMessage.setTopic(userTopic);

    mq = mQClientFactory.getClientConfig().queueWithNamespace(selector.select(messageQueueList, userMessa
} catch (Throwable e) {
    throw new MQClientException("select message queue throwed exception.", e);
}

// 省略部分代码

// 发送消息
if (mq != null) {
    return this.sendKernelImpl(msg, mq, communicationMode, sendCallback, null, timeout - costTime);
} else {
    throw new MQClientException("select message queue return null.", null);
}
// 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sendKernelImpl()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sendKernelImpl()。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，都统一到了同一个流程中。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分了三种发送方式：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

lmtoo

4

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

2019-09-10

微微一笑

2

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~~~

2019-09-11

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。我们讲得如何来实现异步网络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。

2019-09-14

明日

2

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。

2019-09-10

每天晒白牙 1

我总结的kafka生产消息的源码分析
https://mp.weixin.qq.com/s/-s34_y16HU6HR5HDsSD4bg

2019-09-10

leslie 1

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和方案，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。

2019-09-10

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。

2019-09-11

DFighting 0

这里使用异步主要是提升消息发送的吞吐量，而在这过程中影响吞吐的有两个：磁盘IO和网络IO，而现在的这种异步方式好像并没有为这两部分设计详细的优化，似乎只是简单用了一个多线程去执行各自的操作，也就是并没有对真正的性能短板做优化。我觉得真要是异步的话，需要在这里维持一个队列、一个磁盘IO选择器和一个网络发送IO选择器，真正异步的点应该是两个选择器加并发协调处理待发送的数据。不过就像有些同学评价的，netty底层对各种IO已经做了很好的支持，这里的异步就显得很苍白无力了。不知netty是怎么设计异步来达到极致的性能的。

2019-09-16

humor 0

一是处理异常的代码很奇怪吧，有的异常使用sendCallback抛出，有的直接抛出；二是超时的语义有问题，现在的timeout意思是消息在线程池中排队的时间

2019-09-16

墙角儿的花

0

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。

2019-09-11

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 x 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。

2019-09-12

QQ怪

0

老师，节日快乐

2019-09-10

作者回复

感谢！

2019-09-11

约书亚

0

同楼上@lmtoo答案，源码一直追下去发现回调主要还是NettyRemoting做的，回调事件应该发生在netty的event executor绑定的线程内。最上层创建线程池没什么意义。改进的话是不是线程池去掉了就可以了。

2019-09-10

leslie

0

前期一直忙着强化和梳理一些基本功：操作系统、网络这块，学到现在发现老师的课程中的代码能看懂，大致思路也能明白；就是写不出。Python或者Go可以么？ Java实在、、、Python和Go极客时间都有购买课程。

可能目前线上的存储中间件现状比较差【许老师的课程对数据存储的定义，觉得有道理就直接现用了】，尤其是Nosql DB和MQ基本处于闲置，故而一直焦虑在这块；可是当现在初期迷惑已经解除且基本清晰时发现学习这门课和使用MQ的瓶颈就在代码能力上，毕竟DBA的Coding能力都比较差尤其是开发相关的能力；准备开始把之前报的开发语言的课程学习一遍。

今天教师节：愿老师节日快乐，感激老师在授课之余一直如此辛勤的回帖解答我们的困惑；谢谢老师。

2019-09-10

作者回复

感谢！

代码能力这块儿，除了学习，还是多写代码，熟能生巧。

2019-09-11

付永强

0

教师节快乐！

2019-09-10

作者回复

感谢！

2019-09-11

业余草

0

教师节，老师们都辛苦了！

2019-09-10

作者回复

感谢！

2019-09-10

Hurt

0

一定要学java吗 老师

2019-09-10

作者回复

不需要一定会Java，但至少要熟练掌握一门编程语言。

2019-09-10