

第38讲 | 对比Java标准NIO类库，你知道Netty是如何实现更高性能的吗？

2018-08-04 杨晓峰





第38讲 | 对比Java标准NIO类库，你知道Netty是如何实现更高性能的吗？

杨晓峰

- 00:18 / 09:27

今天我会对NIO进行一些补充，在[专栏第11讲](#)中，我们初步接触了Java提供的几种IO机制，作为语言基础类库，Java自身的NIO设计更偏底层，这本无可厚非，但是对于一线的应用开发者，其复杂性、扩展性等方面，就存在一定的局限了。在基础NIO之上，Netty构建了更加易用、高性能的网络框架，广泛应用于互联网、游戏、电信等各种领域。

今天我要问你的问题是，[对比Java标准NIO类库，你知道Netty是如何实现更高性能的吗？](#)

典型回答

单独从性能角度，Netty在基础的NIO等类库之上进行了很多改进，例如：

- 更加优雅的Reactor模式实现、灵活的线程模型、利用EventLoop等创新性的机制，可以非常高效地管理成百上千的Channel。
- 充分利用了Java的Zero-Copy机制，并且从多种角度，“斤斤计较”般的降低内存分配和回收的开销。例如，使用池化的Direct Buffer等技术，在提高IO性能的同时，减少了对象的创建和销毁；利用反射等技术直接操纵SelectionKey，使用数组而不是Java容器等。
- 使用更多本地代码。例如，直接利用JNI调用Open SSL等方式，获得比Java内建SSL引擎更好的性能。
- 在通信协议、序列化等其他角度的优化。

总的来说，Netty并没有Java核心类库那些强烈的通用性、跨平台等各种负担，针对性能等特定目标以及Linux等特定环境，采取了一些极致的优化手段。

考点分析

这是一个比较开放的问题，我给出的回答是个概要性的举例说明。面试官很可能利用这种开放问题作为引子，针对你回答的一个或者多个点，深入探讨你在不同层次上的理解程度。

在面试准备中，兼顾整体性的同时，不要忘记选定个别重点进行深入理解掌握，最好是进行源码层面的深入阅读和实验。如果你希望了解更多从性能角度Netty在编码层面的手段，可以参考Norman在Devboxx上的[分享](#)，其中的很多技巧对于实现极致性能的API有一定借鉴意义，但在一般的业务开发中要谨慎采用。

虽然提到Netty，人们会自然地想到高性能，但是Netty本身的优势不仅仅只有这一个方面。

下面我会侧重两个方面：

- 对Netty进行整体介绍，帮你了解其基本组成。
- 从一个简单的例子开始，对比在[第11讲](#)中基于IO、NIO等标准API的实例，分析它的技术要点，给你提供一个进一步深入学习的思路。

知识扩展

首先，我们从整体了解一下Netty。按照官方定义，它是一个异步的、基于事件Client/Server的网络框架，目标是提供一种简单、快速构建网络应用的方式，同时保证高吞吐量、低延时、高可靠性。

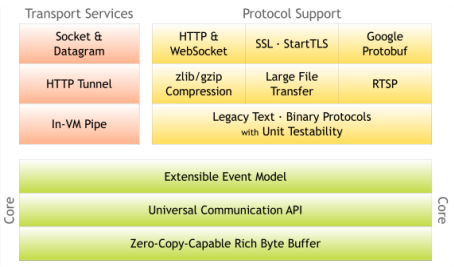
从设计思路和目的上，Netty与Java自身的NIO框架相比有哪些不同呢？

我们知道Java的标准类库，由于其基础性、通用性的定位，往往过于关注技术模型上的抽象，而不是从一线应用开发者的角度去思考。我曾提到过，引入并发包的一个重要原因就是，应用开发者使用Thread API比较痛苦，需要操心的不仅仅是业务逻辑，而且还要自己负责将其映射到Thread模型上。Java NIO的设计也有类似的特点，开发者需要深入掌握线程、IO、网络等相关概念，学习路径很长，很容易导致代码复杂、晦涩，即使是有经验的工程师，也难以快速地写出高可靠性的实现。

Netty的设计强调了“Separation Of Concerns”，通过精巧设计的事件机制，将业务逻辑和无关技术逻辑进行隔离，并通过各种方便的抽象，一定程度上填补了了基础平台和业务开发之间的鸿沟，更有利于在应用开发中普及业界的最佳实践。

另外，Netty > java.nio + java.net！

从API能力范围来看，Netty完全是Java NIO框架的一个大大的超集，你可以参考Netty官方的模块划分。



除了核心的事件机制等，Netty还额外提供了很多功能，例如：

- 从网络协议的角度，Netty除了支持传输层的UDP、TCP、[SCTP](#)协议，也支持HTTP(s)、WebSocket等多种应用层协议，它并不是单一协议的API。
- 在应用中，需要将数据从Java对象转换成为各种应用协议的数据格式，或者进行反向的转换，Netty为此提供了一系列扩展的编解码框架，与应用开发场景无缝衔接，并且性能良好。
- 它扩展了Java NIO Buffer，提供了自己的ByteBuf实现，并且深度支持Direct Buffer等技术，甚至hack了Java内部对Direct Buffer的分配和销毁等。同时，Netty也提供了更加完善的Scatter/Gather机制实现。

可以看到，Netty的能力范围大大超过了Java核心类库中的NIO等API，可以说它是一个从应用视角出发的产物。

当然，对于基础API设计，Netty也有自己独到的见解，未来Java NIO API也可能据此进行一定的改进，如果你有兴趣可以参考[JDK-8187540](#)。

接下来，我们一起来看一个入门的代码实例，看看Netty应用到底是什么样子。

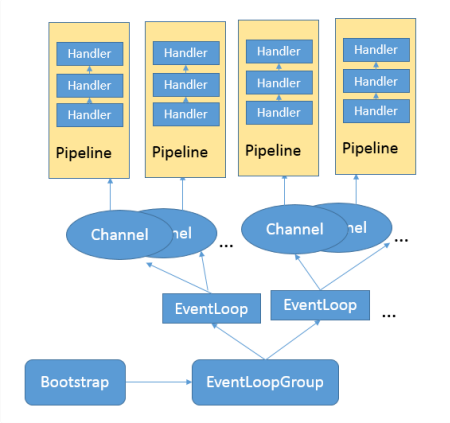
与[第11讲](#)类似，同样是以简化的Echo Server为例，下图是Netty官方提供的Server部分，完整用例请点击[链接](#)。

```
public final class EchoServer {  
    static final boolean SSL = System.getProperty("ssl") != null;  
    static final int PORT = Integer.parseInt(System.getProperty("port", "8007"));  
  
    public static void main(String[] args) throws Exception {  
        // Configure SSL.  
        final SslContext sslCtx;  
        if (SSL) {  
            SelfSignedCertificate ssc = new SelfSignedCertificate();  
            sslCtx = SslContextBuilder.forServer(ssc.certificate(), ssc.privateKey()).build();  
        } else {  
            sslCtx = null;  
        }  
  
        // Configure the server.  
        EventLoopGroup bossGroup = new NioEventLoopGroup(1);  
        EventLoopGroup workerGroup = new NioEventLoopGroup();  
        final EchoServerHandler serverHandler = new EchoServerHandler();  
        try {  
            ServerBootstrap b = new ServerBootstrap();  
            b.group(bossGroup, workerGroup)  
            .channel(NioServerSocketChannel.class)  
            .option(ChannelOption.SO_BACKLOG, 100)  
            .handler(new LoggingHandler(LogLevel.INFO))  
            .childHandler(new ChannelInitializer<SocketChannel>() {  
                @Override  
                public void initChannel(SocketChannel ch) throws Exception {  
                    ChannelPipeline p = ch.pipeline();  
                    if (sslCtx != null) {  
                        p.addLast(sslCtx.newHandler(ch.alloc()));  
                    }  
                    //p.addLast(new LoggingHandler(LogLevel.INFO));  
                    p.addLast(serverHandler);  
                }  
            });  
  
            // Start the server.  
            ChannelFuture f = b.bind(PORT).sync();  
  
            // Wait until the server socket is closed.  
            f.channel().closeFuture().sync();  
        } finally {  
            // Shut down all event loops to terminate all threads.  
            bossGroup.shutdownGracefully();  
            workerGroup.shutdownGracefully();  
        }  
    }  
}
```

上面的例子，虽然代码很短，但已经足够体现出Netty的几个核心概念，请注意我用红框标记出的部分：

- ServerBootstrap**，服务器端程序的入口，这是Netty为简化网络程序配置和关闭等生命周期管理，所引入的Bootstrapping机制。我们通常要做的创建Channel、绑定端口、注册Handler等，都可以通过这个统一的入口，以Fluent API等形式完成，相对简化了API使用。与之相对应，**Bootstrap**则是Client端的通常入口。
- Channel**，作为一个基于NIO的扩展框架，Channel和Selector等概念仍然是Netty的基础组件，但是针对应用开发具体需求，提供了相对易用的抽象。
- EventLoop**，这是Netty处理事件的核心机制。例子中使用了EventLoopGroup。我们在NIO中通常要做的几件事情，如注册感兴趣的事件、调度相应的Handler等，都是EventLoop负责。
- ChannelFuture**，这是Netty实现异步IO的基础之一，保证了同一个Channel操作的调用顺序。Netty扩展了Java标准的Future，提供了针对自己场景的特有Future定义。
- ChannelHandler，这是应用开发者放置业务逻辑的主要地方，也是我上面提到的“Separation Of Concerns”原则的体现。
- ChannelPipeline**，它是ChannelHandler链条的容器，每个Channel在创建后，自动被分配一个ChannelPipeline。在上面的示例中，我们通过ServerBootstrap注册了ChannelInitializer，并且实现了initChannel方法，而在该方法中则承担了向ChannelPipeline安装其他Handler的任务。

你可以参考下面的简化示意图，忽略Inbound/OutBound Handler的细节，理解这几个基本单元之间的操作流程和对应关系。



对比Java标准NIO的代码，Netty提供的相对高层次的封装，减少了对Selector等细节的操纵，而EventLoop、Pipeline等机制则简化了编程模型，开发者不用担心并发等问题，在一定程度上简化了应用代码的开发。最难能可贵的是，这一切并没有以可靠性、可扩展性为代价，反而将其大幅度提高。

我在[专栏周末福利](#)中已经推荐了Norman Maurer等编写的《Netty实战》（Netty In Action），如果你想系统学习Netty，它是个很好的入门参考。针对Netty的一些实现原理，很可能成为面试中的考点，例如：

- Reactor模式和Netty线程模型。
- Pipelining、EventLoop等部分的设计实现细节。
- Netty的内存管理机制、[引用计数](#)等特别手段。
- 有的时候面试官也喜欢对比Java标准NIO API，例如，你是否知道Java NIO早期版本中的Epoll[空转问题](#)，以及Netty的解决方式等。

对于这些知识点，公开的深入解读已经有很多了，在学习时希望你不要一开始就被复杂的细节弄晕，可以结合实例，逐步、有针对性的进行学习。我的一个建议是，可以试着画出相应的示意图，非常有助于理解并能清晰阐述自己的看法。

今天，从Netty性能的问题开始，我概要地介绍了Netty框架，并且以Echo Server为例，对比了Netty和Java NIO在设计上的不同。但这些都仅仅是冰山的一角，全面掌握还需要下非常多的功夫。

一课一练

关于今天我们讨论的题目你做到心中有数了吗？今天的思考题是，Netty的线程模型是什么样的？

请在留言区写写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习奖励礼券，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。



忆水寒

2018-08-06

Netty采用Reactor线程模型。这里面主要有三种Reactor线程模型。分别是单线程模式、主从Reactor模式、多Reactor线程模式。其都可以通过初试和EventLoopGroup进行设置。其主要区别在于，单Reactor模式就是一个线程，既进程处理连接，也处理IO，类似于我们传统的OIO编程。主从Reactor模式，其实就是将监听连接和处理IO的分开在不同的线程完成。最后，主从Reactor线程模型，为了解决多Reactor模型下单一线程性能不足的问题，改为了一组线程池进行处理。官方默认的是采用这种主从Reactor模型。其线程数默认为CPU内核的2倍。杨老师，不知道我说的对不对？

作者回复

2018-08-07

对的

Levy	
netty线程模型一般分为监听线程和I/O处理线程，也即bossGroup和workerGroup，属于多Reactor模型 作者回复	2018-08-04
是的，不同版本模型有点区别，但逻辑上都还是区分	2018-08-05
zt	
老师，能给说下mina和netty的相同和不同吗？ 作者回复	2018-08-06
mina没仔细研究过...	2018-08-07
咖啡猫口里的咖啡猫💎💎	
为什么我看看开源项目把netty做网络层，模型都是1-N，为什么N-N会浪费资源	2018-08-05
L	
杨老师，可以的话麻烦推荐几个能获取到最新Java开发资料，或者编程领域相关的外文网站，博客等，就是想关注一些最新的技术信息	2018-08-05
Allen	
请问为啥不推荐netty权威指南？💎💎	2018-08-05
Allen	
有三种模型。1-1。1-n n-n。并且可以根据实际情况自动进行调整，可谓是线程模型的终极版本，简直也太酷了	2018-08-05



