

架构师

ARCHITECT

| 特刊 |

深入浅出Netty

SPECIAL ISSUE

Oct, 2016

架构师特刊



Geekbang
极客邦科技

InfoQ

序言

最近几年，Netty 社区的发展如火如荼，无论是大数据领域，还是微服务架构，底层都需要一个高效的分布式通信框架作为基础组件。

Netty 凭借优异的性能、灵活的可扩展新得到了广泛的应用。短短几年间，Netty 已经成为众多 Java 高性能异步通信框架的首选。

作为 Java 语言领域最流行、表现最优异的 NIO 框架，Netty 深受大家喜爱，但是长期以来除了 UserGuide 之外，国内鲜有 Netty 相关的

系统性文章供广大 NIO 编程爱好者学习和参考。由于 Netty 源码的复杂性和 NIO 编程本身的技术门槛限制，对于大多数初学者而言，通过

自己阅读和分析源码来深入掌握 Netty 的设计原理和实现细节是件非常困难的事情。

为了方便大家系统性的学习 Netty，2014 年春节前后，我分享了博文

《Netty5.0 架构剖析和源码解读》，短短几个月的时间，阅读和下载量超过 10 万次。很多网友建议我能够继续按照专题的形式分析和解析 Netty 的架构和源码，以及实际应用案例。于是从 2014 年 5 月份开始，我正式在 InfoQ 社区分享 Netty 相关的专题文章，涉及到性能、可靠性、编解码、定制性以及案例剖析等。这些文章深受大家的喜爱，几乎每期都是热点内容排名 TOP5。

2 年多的时间，在 InfoQ 分享的 Netty 专题文章超过 10 篇，通过其它方式也陆续分享了一些 Netty 的实际案例，为了便于大家集中学习，很有必要对这些已经发表的文章进行汇总和提取，形成一本迷你书，奉献给各位读者。

感谢主编郭蕾的帮助和支持，InfoQ 上的 Netty 专题，都是由他亲自策划、编辑和校审的。感谢 InfoQ 这个平台，为广大 Netty 爱好者提供了免费学习和交流的技术乐土。

由于自身水平所限，文章难免存在遗漏或者错误，欢迎广大读者批评指正。

李林峰

目录



- 01** Netty 入门
- 02** Netty 服务端创建
- 03** Netty 客户端创建
- 04** Netty 消息的发送和接收
- 05** Netty 线程模型
- 06** Netty 架构剖析
- 07** Netty 案例集锦

1 Netty 入门

1.1 传统的 BIO 编程

网络编程的基本模型是 Client/Server 模型，也就是两个进程之间进行相互通信，其中服务端提供位置信息（绑定的 IP 地址和监听端口），客户端通过连接操作向服务端监听的地址发起连接请求，通过三次握手建立连接，如果连接建立成功，双方就可以通过网络套接字（Socket）进行通信。

在基于传统同步阻塞模型开发中，ServerSocket 负责绑定 IP 地址，启动监听端口；Socket 负责发起连接操作。连接成功之后，双方通过输入和输出流进行同步阻塞式通信。

1.1.1 BIO 通信模型图

首先，我们通过图 2-1 所示的通信模型图来熟悉下 BIO 的服务端通信模型：采用 BIO 通信模型的服务端，通常由一个独立的 Acceptor 线程负责监听客户端的连接，它接收到客户端连接请求之后为每个客户端创建一个新的线程进行链路处理，处理完成之后，通过输出流返回应答给客户端，线程销毁。这就是典型的一请求一应答通信模型。

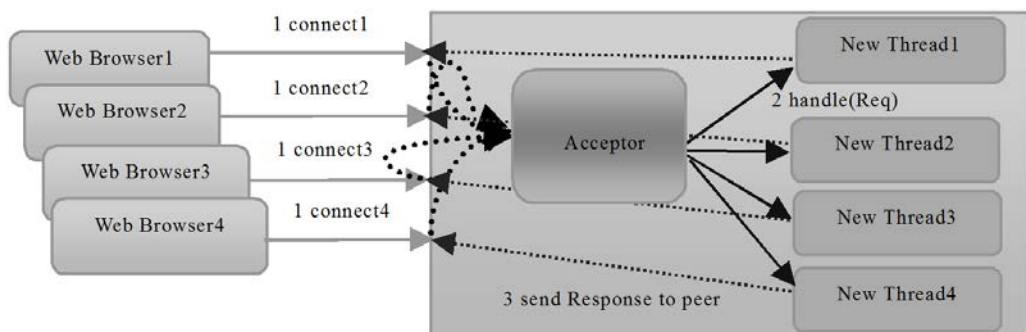


图 1-1 同步阻塞 I/O 服务端通信模型（1 客户端 1 线程）

该模型最大的问题就是缺乏弹性伸缩能力，当客户端并发访问量增加后，服务端的线程个数和客户端并发访问数呈 1:1 的正比关系，由于线程是 Java 虚拟机非常宝贵的系统资源，当线程数膨胀之后，系统的性能将急剧下降，随着并发访问量的继续增大，系统会发生线程堆栈溢出、创建新线程失败等问题，并最终导致进程宕机或者僵死，不能对外提供服务。

1.2 伪异步 I/O 编程

为了解决同步阻塞 I/O 面临的一个链路需要一个线程处理的问题，后来有人对它的线程模型进行了优化，后端通过一个线程池来处理多个客户端的请求接入，形成客户端个数 M：线程池最大线程数 N 的比例关系，其中 M 可以远远大于 N，通过线程池可以灵活的调配线程资源，设置线程的最大值，防止由于海量并发接入导致线程耗尽。

下面，我们结合连接模型图和源码，对伪异步 I/O 进行分析，看它是否能够解决同步阻塞 I/O 面临的问题。

1.2.1 伪异步 I/O 模型图

采用线程池和任务队列可以实现一种叫做伪异步的 I/O 通信框架，它的模型图如图 1-2 所示。

当有新的客户端接入的时候，将客户端的 Socket 封装成一个 Task（该任务实现 `java.lang.Runnable` 接口）投递到后端的线程池中进行处理，JDK 的线程

池维护一个消息队列和 N 个活跃线程对消息队列中的任务进行处理。由于线程池可以设置消息队列的大小和最大线程数，因此，它的资源占用是可控的，无论多少个客户端并发访问，都不会导致资源的耗尽和宕机。

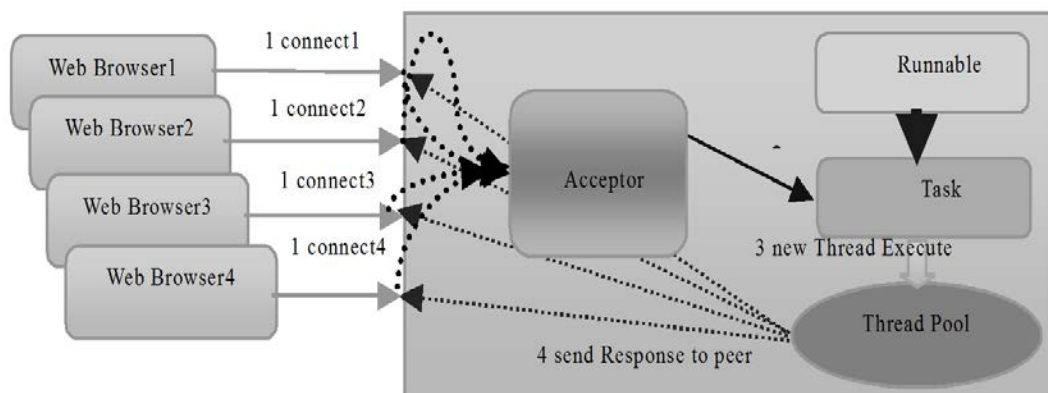


图 1-2 伪异步 I/O 服务端通信模型 (M: N)

伪异步 I/O 实际上仅仅只是对之前 I/O 线程模型的一个简单优化，它无法从根本上解决同步 I/O 导致的通信线程阻塞问题。下面我们就简单分析下如果通信对方返回应答时间过长，会引起的级联故障。

1. 服务端处理缓慢，返回应答消息耗费60s，平时只需要10ms。
2. 采用伪异步I/O的线程正在读取故障服务节点的响应，由于读取输入流是阻塞的，因此，它将会被同步阻塞60s。
3. 假如所有的可用线程都被故障服务器阻塞，那后续所有的I/O消息都将在队列中排队。
4. 由于线程池采用阻塞队列实现，当队列积满之后，后续入队列的操作将被阻塞。
5. 由于前端只有一个Accptor线程接收客户端接入，它被阻塞在线程池的同步阻塞队列之后，新的客户端请求消息将被拒绝，客户端会发生大量的连接超时。
6. 由于几乎所有的连接都超时，调用者会认为系统已经崩溃，无法接收新的

请求消息。

1.3 NIO编程

在介绍 NIO 编程之前，我们首先需要澄清一个概念：NIO 到底是什么的简称？有人称之为 New I/O，因为它相对于之前的 I/O 类库是新增的，所以被称为 New I/O，这是它的官方叫法。但是，由于之前老的 I/O 类库是阻塞 I/O，New I/O 类库的目标就是要让 Java 支持非阻塞 I/O，所以，更多的人喜欢称之为非阻塞 I/O（Non-block I/O），由于非阻塞 I/O 更能够体现 NIO 的特点，所以本文使用的 NIO 都指的是非阻塞 I/O。

与 Socket 类和 ServerSocket 类相对应，NIO 也提供了 SocketChannel 和 ServerSocketChannel 两种不同的套接字通道实现。这两种新增的通道都支持阻塞和非阻塞两种模式。阻塞模式使用非常简单，但是性能和可靠性都不好，非阻塞模式则正好相反。开发人员一般可以根据自己的需要来选择合适的模式，一般来说，低负载、低并发的应用程序可以选择同步阻塞 I/O 以降低编程复杂度，但是对于高负载、高并发的网络应用，需要使用 NIO 的非阻塞模式进行开发。

1.4 AIO编程

NIO2.0 引入了新的异步通道的概念，并提供了异步文件通道和异步套接字通道的实现。异步通道提供两种方式获取操作结果：

- 通过 `java.util.concurrent.Future` 类来表示异步操作的结果；
- 在执行异步操作的时候传入一个 `java.nio.channels`；
- `CompletionHandler` 接口的实现类作为操作完成的回调。

NIO2.0 的异步套接字通道是真正的异步非阻塞 I/O，它对应 UNIX 网络编程中的事件驱动 I/O（AIO），它不需要通过多路复用器（Selector）对注册的通道进行轮询操作即可实现异步读写，从而简化了 NIO 的编程模型。

1.5 几种I/O模型对比

不同的 I/O 模型由于线程模型、API 等差别很大，所以用法的差异也非常大。

由于之前的几个小节已经集中对这几种 I/O 的 API 和用法进行了说明，本小节会重点对这几种 I/O 进行功能对比。如表 2-1 所示。

表 1-1 几种 I/O 模型的功能和特性对比

	同步阻塞I/O（BIO）	伪异步I/O	非阻塞I/O（NIO）	异步I/O（AIO）
客户端个数：I/O线程	1: 1	M: N（其中M可以大于N）	M: 1（1个I/O线程处理多个客户端连接）	M: 0（不需要启动额外的I/O线程，被动回调）
I/O类型（同步）	同步I/O	同步I/O	同步I/O（I/O多路复用）	异步I/O
I/O类型（阻塞）	阻塞I/O	阻塞I/O	非阻塞I/O	非阻塞I/O
API使用难度	简单	简单	非常复杂	复杂
调试难度	简单	简单	复杂	复杂
可靠性	非常差	差	高	高
吞吐量	低	中	高	高

1.6 业界主流的NIO框架介绍

随着移动互联网的发展和大数据时代的到来，大规模分布式服务框架、分布式流计算框架已经成为架构主流，分布式服务节点之间的通信形式往往是内部长连接，例如 FaceBook 的 Thrift 协议，为了提升节点间的通信吞吐量、提升通信性能，目前主流的内部通信框架均使用 NIO 框架，对于大公司、技术积累比较深的团队可能会使用自研的 NIO 框架来满足个性化或者行业特殊的需求，但是大多数架构师会选择业界主流的 NIO 框架进行异步通信开发。

目前，业界主流的 NIO 框架主要有两款：Mina 和 Netty，两者都使用 Apache LICENSE-2.0 进行开源。不同之处是 Mina 是 Apache 基金会的官方 NIO 框架，Netty 之前是 Jboss 的 NIO 框架，后来脱离 Jboss 独立申请了 netty.io 域名，与 Jboss 脱离关系，并对版本进行了重构，导致 API 无法向上兼容。

Mina 和 Netty 还有一段历史渊源，Mina 最初版本的架构师是 Trustin Lee，后来，由于种种原因，Trustin Lee 离开了 Mina 社区加入到了 Netty 团队，重新设计并开发了 Netty。很多读者会发现 Netty 中透着 Mina 的影子，两个框架的架构理念也有很多相似之处，甚至一些代码都非常相似，原因就在这里。

目前，Mina 和 Netty 的应用已经非常广泛，很多开源框架都使用两者做底

层的 NIO 框架，例如 Hadoop 的通信组件 Avro 使用 Netty 做底层的通信框架，Openfire 则使用 Mina 做底层通信框架，相比于 Mina，Netty 社区目前更活跃，版本应用范围也更广。

1.7 为什么选择Netty

1.7.1 不选择 Java 原生 NIO 编程的原因

现在我们总结一下为什么不建议开发者直接使用 JDK 的 NIO 类库进行开发，具体原因如下。

1. NIO 的类库和 API 繁杂，使用麻烦，你需要熟练掌握 Selector、ServerSocketChannel、SocketChannel、ByteBuffer 等。
2. 需要具备其他的额外技能做铺垫，例如熟悉 Java 多线程编程。这是因为 NIO 编程涉及到 Reactor 模式，你必须对多线程和网络编程非常熟悉，才能编写出高质量的 NIO 程序。
3. 可靠性能力补齐，工作量和难度都非常大。例如客户端面临断连重连、网络闪断、半包读写、失败缓存、网络拥塞和异常码流的处理等问题，NIO 编程的特点是功能开发相对容易，但是可靠性能力补齐的工作量和难度都非常大。
4. JDK NIO 的 BUG，例如臭名昭著的 epoll bug，它会导致 Selector 空轮询，最终导致 CPU 100%。官方声称在 JDK 1.6 版本的 update18 修复了该问题，但是直到 JDK 1.7 版本该问题仍旧存在，只不过该 BUG 发生概率降低了一些而已，它并没有被根本解决。该 BUG 以及与该 BUG 相关的问题单可以参见以下链接内容。

- http://bugs.java.com/bugdatabase/view_bug.do?bug_id=6403933
- http://bugs.java.com/bugdatabase/view_bug.do?bug_id=2147719

由于上述原因，在大多数场景下，不建议大家直接使用 JDK 的 NIO 类库，除非你精通 NIO 编程或者有特殊的需求。在绝大多数的业务场景中，我们可以使用

NIO 框架 Netty 来进行 NIO 编程，它既可以作为客户端也可以作为服务端，同时支持 UDP 和异步文件传输，功能非常强大。

1.7.2 选择 Netty 的理由

Netty 是业界最流行的 NIO 框架之一，它的健壮性、功能、性能、可定制性和可扩展性在同类框架中都是首屈一指的，它已经得到成百上千的商用项目验证，例如 Hadoop 的 RPC 框架 avro 使用 Netty 作为底层通信框架；很多其他业界主流的 RPC 框架，也使用 Netty 来构建高性能的异步通信能力。

通过对 Netty 的分析，我们将它的优点总结如下：

- API使用简单，开发门槛低；
- 功能强大，预置了多种编解码功能，支持多种主流协议；
- 定制能力强，可以通过ChannelHandler对通信框架进行灵活地扩展；
- 性能高，通过与其他业界主流的NIO框架对比，Netty的综合性能最优；
- 成熟、稳定，Netty修复了已经发现的所有JDK NIO BUG，业务开发人员不需要再为NIO的BUG而烦恼；
- 社区活跃，版本迭代周期短，发现的BUG可以被及时修复，同时，更多的新功能会加入；
- 经历了大规模的商业应用考验，质量得到验证。在互联网、大数据、网络游戏、企业应用、电信软件等众多行业得到成功商用，证明了它已经完全能够满足不同行业的商业应用了。

正是因为这些优点，Netty 逐渐成为 Java NIO 编程的首选框架。

Netty 的架构图如下所示。

1.8 Netty开发环境搭建

首先假设你已经在本机安装了 JDK1.7，配置了 JDK 的环境变量 path，同时下载并正确启动了 IDE 工具 Eclipse。如果你是个 Java 初学者，从来没有在本机搭建过 Java 开发环境，建议你先选择一本 Java 基础入门的书籍或者课程学习。

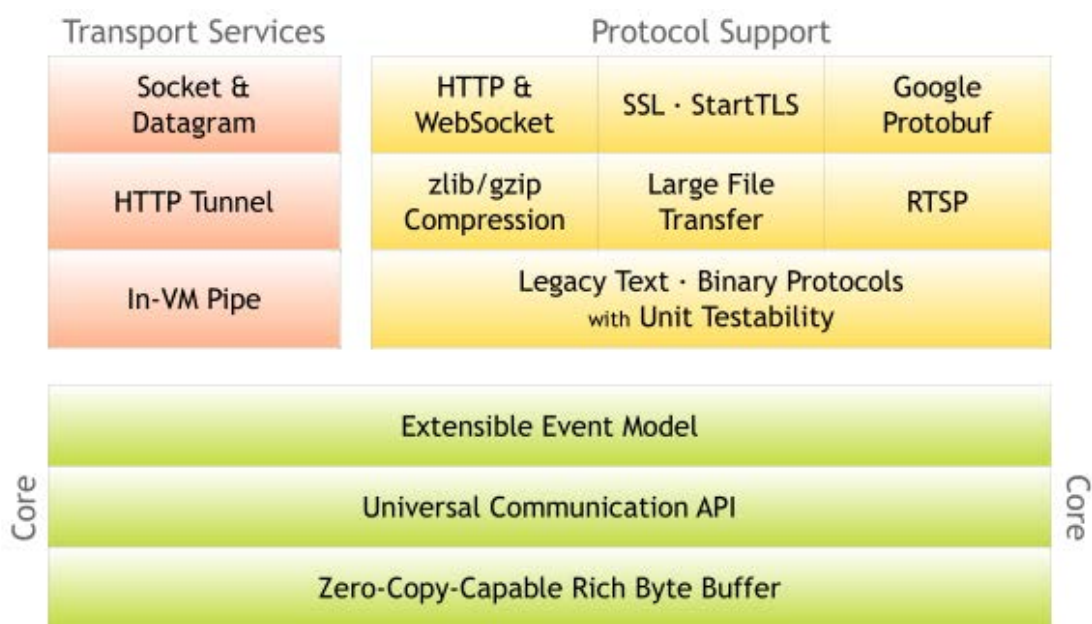


图 1-3 Netty 架构图

假如你习惯于使用其他 IDE 工具进行 Java 开发，例如 NetBeans IDE，也可以运行本节的入门例程。但是，你需要根据自己实际使用的 IDE 进行对应的配置修改和调整，本书统一使用 eclipse-jee-kepler-SR1-win32 作为 Java 开发工具。

1.8.1 下载 Netty 类库

访问 Netty 的官网 <http://netty.io/>，从【Downloads】标签页选择下载 4.1.5.Final 软件包，包含了源码、编译类库和 Java Doc，18.1M 左右，解压之后的软件包如下所示。

这时会发现里面包含了各个模块的 .jar 包和源码，由于我们直接以二进制类库的方式使用 Netty，所以只需要获取 netty-all-4.1.5.Final.jar 即可。

1.8.2 开发工程搭建

将 netty-all-4.1.5.Final.jar 导入到 Java 工程的 lib 目录下（lib 目录需要自建），右键单击 netty-all-4.1.5.Final.jar，在弹出的菜单中，选择将 .jar 包添加到 Build Path 中，即可完成 Netty 开发环境的搭建。





















 all-in-one		
 netty-buffer-4.1.5.Final.jar	247.3 KB	247.5 KB
 netty-buffer-4.1.5.Final-sources.jar	155.6 KB	156.0 KB
 netty-build-22.jar	14.1 KB	14.5 KB
 netty-build-22-sources.jar	8.2 KB	8.5 KB
 netty-codec-4.1.5.Final.jar	297.4 KB	297.5 KB
 netty-codec-4.1.5.Final-sources.jar	246.1 KB	246.5 KB
 netty-codec-http-4.1.5.Final.jar	524.5 KB	525.0 KB
 netty-codec-http-4.1.5.Final-sources.jar	409.2 KB	409.5 KB
 netty-codec-http2-4.1.5.Final.jar	333.4 KB	333.5 KB
 netty-codec-http2-4.1.5.Final-sources.jar	222.0 KB	222.0 KB
 netty-codec-memcache-4.1.5.Final.jar	41.8 KB	42.0 KB
 netty-codec-memcache-4.1.5.Final-sources.jar	39.7 KB	40.0 KB
 netty-codec-redis-4.1.5.Final.jar	40.1 KB	40.5 KB
 netty-codec-redis-4.1.5.Final-sources.jar	30.7 KB	31.0 KB
 netty-codec-socks-4.1.5.Final.jar	117.3 KB	117.5 KB
 netty-codec-socks-4.1.5.Final-sources.jar	84.8 KB	85.0 KB
 netty-codec-stomp-4.1.5.Final.jar	25.3 KB	25.5 KB
 netty-codec-stomp-4.1.5.Final-sources.jar	20.4 KB	20.5 KB
 netty-common-4.1.5.Final.jar	669.6 KB	670.0 KB

图 1-4 Netty 软件包

2 Netty 服务端创建

当我们直接使用 JDK NIO 的类库开发基于 NIO 的异步服务端时，需要使用到多路复用器 Selector、ServerSocketChannel、SocketChannel、ByteBuffer、SelectionKey 等等，相比于传统的 BIO 开发，NIO 的开发要复杂很多，开发出稳定、高性能的异步通信框架，一直是个难题。

Netty 为了向使用者屏蔽 NIO 通信的底层细节，在和用户交互的边界做了封装，目的就是减少用户开发工作量，降低开发难度。ServerBootstrap 是 Socket 服务端的启动辅助类，用户通过 ServerBootstrap 可以方便的创建 Netty 的服务端。

2.1 Netty服务端创建时序图

步骤 1：创建 ServerBootstrap 实例。ServerBootstrap 是 Netty 服务端的启动辅助类，它提供了一系列的方法用于设置服务端启动相关的参数。底层通过门面模式对各种能力进行抽象和封装，尽量不需要用户跟过多的底层 API 打交道，降低用户的开发难度。

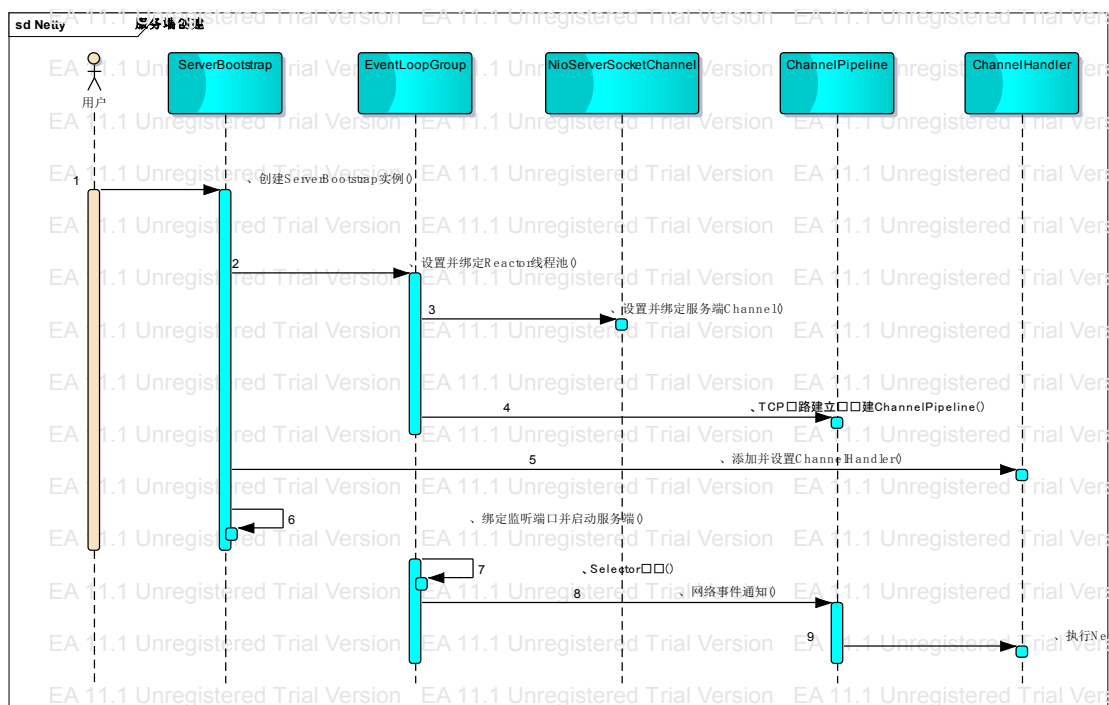


图 2-1 Netty 服务端创建时序图

我们在创建 `ServerBootstrap` 实例时，会惊讶的发现 `ServerBootstrap` 只有一个无参的构造函数，作为启动辅助类这让人不可思议，因为它需要与多个其它组件或者类交互。`ServerBootstrap` 构造函数没有参数的根本原因是因为它的参数太多了，而且未来也可能会发生变化，为了解决这个问题，就需要引入 Builder 模式。《Effective Java》第二版第 2 条建议遇到多个构造器参数时要考虑用构建器，关于多个参数构造函数的缺点和使用构建器的优点大家可以查阅《Effective Java》，在此不再详述。

步骤 2：设置并绑定 Reactor 线程池。Netty 的 Reactor 线程池是 `EventLoopGroup`，它实际就是 `EventLoop` 的数组。`EventLoop` 的职责是处理所有注册到本线程多路复用器 `Selector` 上的 `Channel`，`Selector` 的轮询操作由绑定的 `EventLoop` 线程 `run` 方法驱动，在一个循环体内循环执行。值得说明的是，`EventLoop` 的职责不仅仅是处理网络 I/O 事件，用户自定义的 `Task` 和定时任务 `Task` 也统一由 `EventLoop` 负责处理，这样线程模型就实现了统一。从调度层面看，

也不存在在 EventLoop 线程中再启动其它类型的线程用于异步执行其它的任务，这样就避免了多线程并发操作和锁竞争，提升了 I/O 线程的处理和调度性能。

步骤 3：设置并绑定服务端 Channel。作为 NIO 服务端，需要创建 ServerSocketChannel, Netty 对原生的 NIO 类库进行了封装，对应实现是 NioServerSocketChannel。对于用户而言，不需要关心服务端 Channel 的底层实现细节和工作原理，只需要指定具体使用哪种服务端 Channel 即可。因此，Netty 的 ServerBootstrap 方法提供了 channel 方法用于指定服务端 Channel 的类型。Netty 通过工厂类，利用反射创建 NioServerSocketChannel 对象。由于服务端监听端口往往只需要在系统启动时才会调用，因此反射对性能的影响并不大。相关代码如下所示：

```
/**
 * The {@link Class} which is used to create {@link Channel} instances from.
 * You either use this or {@link #channelFactory(ServerChannelFactory)} if your
 * {@link Channel} implementation has no no-args constructor.
 */
public ServerBootstrap channel(Class<? extends ServerChannel> channelClass) {
    if (channelClass == null) {
        throw new NullPointerException("channelClass");
    }
    return channelFactory(new ServerBootstrapChannelFactory<ServerChannel>(channelClass));
}
```

步骤 4：链路建立的时候创建并初始化 ChannelPipeline。ChannelPipeline 并不是 NIO 服务端必需的，它本质就是一个负责处理网络事件的职责链，负责管理和执行 ChannelHandler。网络事件以事件流的形式在 ChannelPipeline 中流转，由 ChannelPipeline 根据 ChannelHandler 的执行策略调度 ChannelHandler 的执行。典型的网络事件如下：

1. 链路注册；
2. 链路激活；
3. 链路断开；
4. 接收到请求消息；
5. 请求消息接收并处理完毕；

6. 发送应答消息;
7. 链路发生异常;
8. 发生用户自定义事件。

步骤 5: 初始化 ChannelPipeline 完成之后, 添加并设置 ChannelHandler。

ChannelHandler 是 Netty 提供给用户定制和扩展的关键接口。利用 ChannelHandler 用户可以完成大多数的功能定制, 例如消息编解码、心跳、安全认证、TSL/SSL 认证、流量控制和流量整形等。Netty 同时也提供了大量的系统 ChannelHandler 供用户使用, 比较实用的系统 ChannelHandler 总结如下:

1. 系统编解码框架-ByteToMessageCodec;
2. 通用基于长度的半包解码器-LengthFieldBasedFrameDecoder;
3. 码流日志打印Handler-LoggingHandler;
4. SSL安全认证Handler-SslHandler;
5. 链路空闲检测Handler-IdleStateHandler;
6. 流量整形Handler-ChannelTrafficShapingHandler;
7. Base64编解码-Base64Decoder和Base64Encoder。

创建和添加 ChannelHandler 的代码示例如下:

```
.childHandler(new ChannelInitializer<SocketChannel>() {
    @Override
    public void initChannel(SocketChannel ch) throws Exception {
        ch.pipeline().addLast(
            //new LoggingHandler(LogLevel.INFO),
            new EchoServerHandler());
    }
});
```

步骤 6: 绑定并启动监听端口。在绑定监听端口之前系统会做一系列的初始化和检测工作, 完成之后, 会启动监听端口, 并将 ServerSocketChannel 注册到 Selector 上监听客户端连接, 相关代码如下:

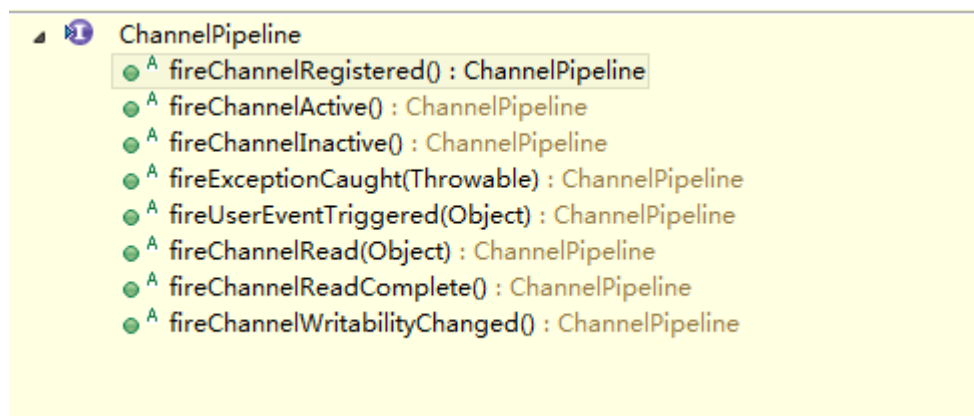
```
@Override
protected void doBind(SocketAddress localAddress) throws Exception {
    javaChannel().socket().bind(localAddress, config.getBacklog());
}
```

步骤 7: Selector 轮询。由 Reactor 线程 NioEventLoop 负责调度和执行 Selector 轮询操作, 选择准备就绪的 Channel 集合, 相关代码如下:

```
private void select() throws IOException {
    Selector selector = this.selector;
    try {
        int selectCnt = 0;
        long currentTimeNanos = System.nanoTime();
        long selectDeadlineNanos = currentTimeNanos + delayNanos(currentTimeNanos);
        for (;;) {
            long timeoutMillis = (selectDeadlineNanos - currentTimeNanos + 500000L) / 1000000L;
            if (timeoutMillis <= 0) {
                if (selectCnt == 0) {
                    selector.selectNow();
                    selectCnt = 1;
                }
                break;
            }
        }

        int selectedKeys = selector.select(timeoutMillis);
        selectCnt ++;
    }
}
```

步骤 8: 当轮询到准备就绪的 Channel 之后, 就由 Reactor 线程 NioEventLoop 执行 ChannelPipeline 的相应方法, 最终调度并执行 ChannelHandler, 代码如下:



步骤 9: 执行 Netty 系统 ChannelHandler 和用户添加定制的 ChannelHandler。ChannelPipeline 根据网络事件的类型, 调度并执行 ChannelHandler, 相关代码如下所示:

```
@Override
public ChannelHandlerContext fireChannelRead(Object msg) {
    DefaultChannelHandlerContext next = findContextInbound(MASK_CHANNEL_READ);
    next.invokeChannelRead(next, msg);
    return this;
}
```

2.2 Netty服务端创建源码分析

首先通过构造函数创建 `ServerBootstrap` 实例，随后，通常会创建两个 `EventLoopGroup`（并不是必须要创建两个不同的 `EventLoopGroup`，也可以只创建一个并共享），代码如下所示：

```
// Configure the server.
EventLoopGroup bossGroup = new NioEventLoopGroup();
EventLoopGroup workerGroup = new NioEventLoopGroup();
```

`NioEventLoopGroup` 实际就是 `Reactor` 线程池，负责调度和执行客户端的接入、网络读写事件的处理、用户自定义任务和定时任务的执行。通过 `ServerBootstrap` 的 `group` 方法将两个 `EventLoopGroup` 实例传入，代码如下：

```
/**
 * Set the {@link EventExecutorGroup} for the parent (acceptor) and the child (client). These
 * {@link EventExecutorGroup}'s are used to handle all the events and IO for {@link SocketChannel} and
 * {@link Channel}'s.
 */
public ServerBootstrap group(EventLoopGroup parentGroup, EventLoopGroup childGroup) {
    super.group(parentGroup);
    if (childGroup == null) {
        throw new NullPointerException("childGroup");
    }
    if (this.childGroup != null) {
        throw new IllegalStateException("childGroup set already");
    }
    this.childGroup = childGroup;
    return this;
}
```

其中父 `NioEventLoopGroup` 被传入了父类构造函数中：

```
/**
 * The {@link EventLoopGroup} which is used to handle all the events for the to-be-created
 */
@SuppressWarnings("unchecked")
public B group(EventLoopGroup group) {
    if (group == null) {
        throw new NullPointerException("group");
    }
    if (this.group != null) {
        throw new IllegalStateException("group set already");
    }
    this.group = group;
    return (B) this;
}
```

该方法会被客户端和服务端重用，用于执行和调度网络事件的读写。

线程组和线程类型设置完成后，需要设置服务端 `Channel`，Netty 通过 `Channel` 工厂类来创建不同类型的 `Channel`，对于服务端，需要创建

NioServerSocketChannel, 所以, 通过指定 Channel 类型的方式创建 Channel 工厂。ServerBootstrapChannelFactory 是 ServerBootstrap 的内部静态类, 职责是根据 Channel 的类型通过反射创建 Channel 的实例, 服务端需要创建的是 NioServerSocketChannel 实例, 代码如下:

```
ServerBootstrapChannelFactory(Class<? extends T> clazz) {
    this.clazz = clazz;
}

@Override
public T newChannel(EventLoop eventLoop, EventLoopGroup childGroup) {
    try {
        Constructor<? extends T> constructor = clazz.getConstructor(EventLoop.class, EventLoopGroup.class);
        return constructor.newInstance(eventLoop, childGroup);
    } catch (Throwable t) {
        throw new ChannelException("Unable to create Channel from class " + clazz, t);
    }
}
```

指定 NioServerSocketChannel 后, 需要设置 TCP 的一些参数, 作为服务端, 主要是要设置 TCP 的 backlog 参数, 底层 C 的对应接口定义如下:

```
int listen(int fd, int backlog);
```

backlog 指定了内核为此套接口排队的最大连接个数, 对于给定的监听套接口, 内核要维护两个队列, 未链接队列和已连接队列, 根据 TCP 三路握手过程中三个分节来分隔这两个队列。backlog 被规定为两个队列总和的最大值, 大多数实现默认值为 5, 但在高并发 web 服务器中此值显然不够, lighttpd 中此值达到 128*8。需要设置此值更大一些的原因是未完成连接队列的长度可能因为客户端 SYN 的到达及等待三路握手第三个分节的到达延时而增大。Netty 默认的 backlog 为 100, 当然, 用户可以修改默认值, 用户需要根据实际场景和网络状况进行灵活设置。

TCP 参数设置完成后, 用户可以为启动辅助类和其父类分别指定 Handler, 两类 Handler 的用途不同, 子类中的 Hanlder 是 NioServerSocketChannel 对应的 ChannelPipeline 的 Handler, 父类中的 Hanlder 是客户端新接入的连接 SocketChannel 对应的 ChannelPipeline 的 Handler。两者的区别可以通过下图来展示:

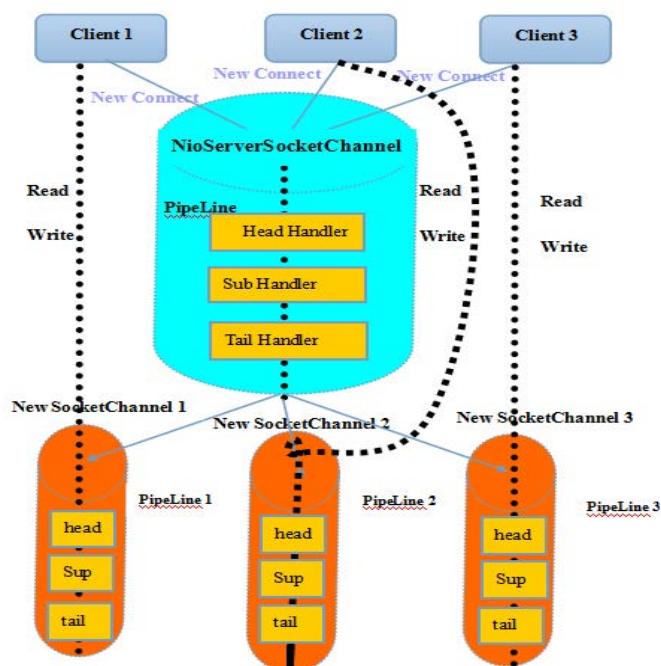


图 2-2 ServerBootstrap 的 Handler 模型

本质区别就是：ServerBootstrap 中的 Handler 是 NioServerSocketChannel 使用的，所有连接该监听端口的客户端都会执行它，父类 AbstractBootstrap 中的 Handler 是个工厂类，它为每个新接入的客户端都创建一个新的 Handler。

服务端启动的最后一步，就是绑定本地端口，启动服务，下面我们来分析下这部分代码：

```
private ChannelFuture doBind(final SocketAddress localAddress) {
    final ChannelFuture regFuture = initAndRegister();
    final Channel channel = regFuture.channel();
    if (regFuture.cause() != null) {
        return regFuture;
    }

    final ChannelPromise promise;
    if (regFuture.isDone()) {
        promise = channel.newPromise();
        doBind0(regFuture, channel, localAddress, promise);
    } else {
        // Registration future is almost always fulfilled already, but just in case it's not.
        promise = new DefaultChannelPromise(channel, GlobalEventExecutor.INSTANCE);
        regFuture.addListener(new ChannelFutureListener() {
            @Override
            public void operationComplete(ChannelFuture future) throws Exception {
                doBind0(regFuture, channel, localAddress, promise);
            }
        });
    }

    return promise;
}
```

先看下 NO.1，首先创建 Channel，createChannel 由子类 ServerBootstrap 实现，创建新的 NioServerSocketChannel，它有两个参数，参数 1 是从父类的 NIO 线程池中顺序获取一个 NioEventLoop，它就是服务端用于监听和接收客户端连接的 Reactor 线程。第二个参数就是所谓的 workerGroup 线程池，它就是处理 IO 读写的 Reactor 线程组，相关代码如下：

```
final ChannelFuture initAndRegister() {
    Channel channel;
    try {
        channel = createChannel();
    } catch (Throwable t) {
        return VoidChannel.INSTANCE.newFailedFuture(t);
    }

    try {
        init(channel);
    } catch (Throwable t) {
        channel.unsafe().closeForcibly();
        return channel.newFailedFuture(t);
    }

    ChannelPromise regFuture = channel.newPromise();
    channel.unsafe().register(regFuture);
    if (regFuture.cause() != null) {
        if (channel.isRegistered()) {
            channel.close();
        } else {
            channel.unsafe().closeForcibly();
        }
    }
}
```

NioServerSocketChannel 创建成功后对它进行初始化，初始化工作主要有三点。

设置 Socket 参数和 NioServerSocketChannel 的附加属性，代码如下：

```
void init(Channel channel) throws Exception {
    final Map<ChannelOption<?>, Object> options = options();
    synchronized (options) {
        channel.config().setOptions(options);
    }

    final Map<AttributeKey<?>, Object> attrs = attrs();
    synchronized (attrs) {
        for (Entry<AttributeKey<?>, Object> e: attrs.entrySet()) {
            @SuppressWarnings("unchecked")
            AttributeKey<Object> key = (AttributeKey<Object>) e.getKey();
            channel.attr(key).set(e.getValue());
        }
    }
}
```

将 AbstractBootstrap 的 Handler 添加到 NioServerSocketChannel 的 ChannelPipeline 中，代码如下：

```
ChannelPipeline p = channel.pipeline();
if (handler() != null) {
    p.addLast(handler());
}
```

将用于服务端注册的 Handler ServerBootstrapAcceptor 添加到 ChannelPipeline 中，代码如下：

```
p.addLast(new ChannelInitializer<Channel>() {
    @Override
    public void initChannel(Channel ch) throws Exception {
        ch.pipeline().addLast(new ServerBootstrapAcceptor(currentChildHandler, currentChildOptions,
            currentChildAttrs));
    }
});
```

到此处，Netty 服务端监听的相关资源已经初始化完毕，就剩下最后一步 - 注册 NioServerSocketChannel 到 Reactor 线程的多路复用器上，然后轮询客户端连接事件。在分析注册代码之前，我们先通过下图看看目前 NioServerSocketChannel 的 ChannelPipeline 的组成：

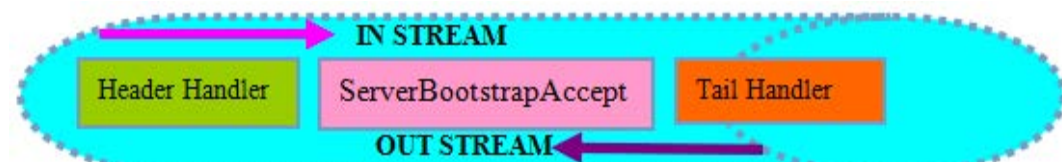


图 2-3 NioServerSocketChannel 的 ChannelPipeline

最后，我们看下 NioServerSocketChannel 的注册。当 NioServerSocketChannel 初始化完成之后，需要将它注册到 Reactor 线程的多路复用器上监听新客户端的接入，代码如下：

首先判断是否是 NioEventLoop 自身发起的操作，如果是，则不存在并发操作，直接执行 Channel 注册；如果由其它线程发起，则封装成一个 Task 放入消息队列中异步执行。此处，由于是由 ServerBootstrap 所在线程执行的注册操作，所以会将其封装成 Task 投递到 NioEventLoop 中执行，代码如下：

```

@Override
public final void register(final ChannelPromise promise) {
    if (eventLoop.inEventLoop()) {
        register0(promise);
    } else {
        try {
            eventLoop.execute(new Runnable() {
                @Override
                public void run() {
                    register0(promise);
                }
            });
        } catch (Throwable t) {
            logger.warn(
                "Force-closing a channel whose registration task was not accepted by an event loop: {}",
                AbstractChannel.this, t);
            closeForcibly();
            closeFuture.setClosed();
            promise.setFailure(t);
        }
    }
}

private void register0(ChannelPromise promise) {
    try {
        // check if the channel is still open as it could be closed in the mean time when the register
        // call was outside of the eventLoop
        if (!ensureOpen(promise)) {
            return;
        }
        doRegister();
        registered = true;
        promise.setSuccess();
        pipeline.fireChannelRegistered();
        if (isActive()) {
            pipeline.fireChannelActive();
        }
    } catch (Throwable t) {
        // Close the channel directly to avoid FD leak.
        closeForcibly();
        closeFuture.setClosed();
        if (!promise.tryFailure(t)) {
            logger.warn(
                "Tried to fail the registration promise, but it is complete already. " +
                "Swallowing the cause of the registration failure:", t);
        }
    }
}
}

```

将NioServerSocketChannel注册到NioEventLoop的Selector上,代码如下:

```

@Override
protected void doRegister() throws Exception {
    boolean selected = false;
    for (;;) {
        try {
            selectionKey = javaChannel().register(eventLoop().selector, 0, this);
            return;
        } catch (CancelledKeyException e) {
            if (!selected) {
                // Force the Selector to select now as the "canceled" SelectionKey may still be
                // cached and not removed because no Select.select(..) operation was called yet.
                eventLoop().selectNow();
                selected = true;
            } else {
                // We forced a select operation on the selector before but the SelectionKey is still cac
                // for whatever reason. JDK bug ?
                throw e;
            }
        }
    }
}
}

```

大伙儿可能会很诧异，应该注册 OP_ACCEPT（16）到多路复用器上，怎么注册 0 呢？0 表示只注册，不监听任何网络操作。这样做的原因如下：

注册方法是多态的，它既可以被 NioServerSocketChannel 用来监听客户端的连接接入，也可以用来注册 SocketChannel，用来监听网络读或者写操作；

通过 SelectionKey 的 interestOps(int ops) 方法可以方便的修改监听操作位。所以，此处注册需要获取 SelectionKey 并给 AbstractNioChannel 的成员变量 selectionKey 赋值。

注册成功之后，触发 ChannelRegistered 事件，方法如下：

```
doRegister();
registered = true;
promise.setSuccess();
pipeline.fireChannelRegistered();
```

Netty 的 HeadHandler 不需要处理 ChannelRegistered 事件，所以，直接调用下一个 Handler，代码如下：

```
@Override
public ChannelHandlerContext fireChannelRegistered() {
    DefaultChannelHandlerContext next = findContextInbound(MASK_CHANNEL_REGISTERED);
    next.invoker().invokeChannelRegistered(next);
    return this;
}
```

当 ChannelRegistered 事件传递到 TailHandler 后结束，TailHandler 也不关心 ChannelRegistered 事件，因此是空实现，代码如下：

```
@Override
public void channelRegistered(ChannelHandlerContext ctx) throws Exception { }
```

ChannelRegistered 事件传递完成后，判断 ServerSocketChannel 监听是否成功，如果成功，需要出发 NioServerSocketChannel 的 ChannelActive 事件，代码如下：

```
if (isActive()) {
    pipeline.fireChannelActive();
}
```

isActive() 也是个多态方法，如果是服务端，判断监听是否启动，如果是客户端，判断 TCP 连接是否完成。ChannelActive 事件在 ChannelPipeline 中传递，完成之后根据配置决定是否自动触发 Channel 的读操作，代码如下：

```

@Override
public ChannelPipeline fireChannelActive() {
    head.fireChannelActive();

    if (channel.config().isAutoRead()) {
        channel.read();
    }

    return this;
}

```

AbstractChannel 的读操作触发 ChannelPipeline 的读操作，最终调用到 HeadHandler 的读方法，代码如下：

```

@Override
public void read(ChannelHandlerContext ctx) {
    unsafe.beginRead();
}

```

继续看 AbstractUnsafe 的 beginRead 方法，代码如下：

```

@Override
public void beginRead() {
    if (!isActive()) {
        return;
    }

    try {
        doBeginRead();
    } catch (final Exception e) {
        invokeLater(new Runnable() {
            @Override
            public void run() {
                pipeline.fireExceptionCaught(e);
            }
        });
        close(voidPromise());
    }
}

```

由于不同类型的 Channel 对读操作的准备工作不同，因此，beginRead 也是个多态方法，对于 NIO 通信，无论是客户端还是服务端，都是要修改网络监听操作位为自身感兴趣的，对于 NioServerSocketChannel 感兴趣的操作是 OP_ACCEPT (16)，于是重新修改注册的操作位为 OP_ACCEPT，代码如下：

在某些场景下，当前监听的操作类型和 Channel 关心的网络事件是一致的，不需要重复注册，所以增加了 & 操作的判断，只有两者不一致，才需要重新注册操作位。

JDK SelectionKey 有四种操作类型，分别为：


```

@Override
protected void doBeginRead() throws Exception {
    if (inputShutdown) {
        return;
    }

    final SelectionKey selectionKey = this.selectionKey;
    if (!selectionKey.isValid()) {
        return;
    }

    final int interestOps = selectionKey.interestOps();
    if ((interestOps & readInterestOp) == 0) {
        selectionKey.interestOps(interestOps | readInterestOp);
    }
}

```

- OP_READ = 1 << 0;
- OP_WRITE = 1 << 2;
- OP_CONNECT = 1 << 3;
- OP_ACCEPT = 1 << 4。

由于只有四种网络操作类型，所以用 4 bit 就可以表示所有的网络操作位，由于 JAVA 语言没有 bit 类型，所以使用了整形来表示，每个操作位代表一种网络操作类型，分别为：0001、0010、0100、1000，这样做的好处是可以非常方便的通过位操作来进行网络操作位的状态判断和状态修改，提升操作性能。

由于创建 NioServerSocketChannel 将 readInterestOp 设置成了 OP_ACCEPT，所以，在服务端链路注册成功之后重新将操作位设置为监听客户端的网络连接操作，初始化 NioServerSocketChannel 的代码如下：

```

/**
 * Create a new instance
 */
public NioServerSocketChannel(EventLoop eventLoop, EventLoopGroup childGroup) {
    super(null, eventLoop, childGroup, newSocket(), SelectionKey.OP_ACCEPT);
    config = new DefaultServerSocketChannelConfig(this, javaChannel().socket());
}

```

到此，服务端监听启动部分源码已经分析完成，接下来，让我们继续分析一个新的客户端是如何接入的。

2.3 客户端接入源码分析

负责处理网络读写、连接和客户端请求接入的 Reactor 线程就是 NioEventLoop，下面我们分析下 NioEventLoop 是如何处理新的客户端连接接入的。当多路复用器检测到新的准备就绪的 Channel 时，默认执行 processSelectedKeysOptimized 方法，代码如下：

```
if (selectedKeys != null) {
    processSelectedKeysOptimized(selectedKeys.flip());
} else {
    processSelectedKeysPlain(selector.selectedKeys());
}
```

由于 Channel 的 Attachment 是 NioServerSocketChannel，所以执行 processSelectedKey 方法，根据就绪的操作位，执行不同的操作，此处，由于监听的是连接操作，所以执行 unsafe.read() 方法，由于不同的 Channel 执行不同的操作，所以 NioUnsafe 被设计成接口，由不同的 Channel 内部的 NioUnsafe 实现类负责具体实现，我们发现 read() 方法的实现有两个，分别是 NioByteUnsafe 和 NioMessageUnsafe，对于 NioServerSocketChannel，它使用的是 NioMessageUnsafe，它的 read 方法代码如下：

```
@Override
public void read() {
    assert eventLoop().inEventLoop();
    if (!config().isAutoRead()) {
        removeReadOp();
    }

    final ChannelConfig config = config();
    final int maxMessagesPerRead = config.getMaxMessagesPerRead();
    final boolean autoRead = config.isAutoRead();
    final ChannelPipeline pipeline = pipeline();
    boolean closed = false;
    Throwable exception = null;
    try {
        for (;;) {
            int localRead = doReadMessages(readBuf);
            if (localRead == 0) {
                break;
            }
            if (localRead < 0) {
                closed = true;
                break;
            }
        }
    }
```

对 doReadMessages 方法进行分析，发现它实际就是接收新的客户端连接并创建 NioSocketChannel 代码如下：

```
@Override
protected int doReadMessages(List<Object> buf) throws Exception {
    SocketChannel ch = javaChannel().accept();

    try {
        if (ch != null) {
            buf.add(new NioSocketChannel(this, childEventLoopGroup().next(), ch));
            return 1;
        }
    } catch (Throwable t) {
        Logger.warn("Failed to create a new channel from an accepted socket.", t);

        try {
            ch.close();
        } catch (Throwable t2) {
            Logger.warn("Failed to close a socket.", t2);
        }
    }

    return 0;
}
```

接收到新的客户端连接后，触发 ChannelPipeline 的 ChannelRead 方法，代码如下：

```
int size = readBuf.size();
for (int i = 0; i < size; i++) {
    pipeline.fireChannelRead(readBuf.get(i));
}
```

执行 headChannelHandlerContext 的 fireChannelRead 方法，事件在 ChannelPipeline 中传递，执行 ServerBootstrapAcceptor 的 channelRead 方法，代码如下。

该方法包含三个主要步骤：

第一步：将启动时传入的 childHandler 加入到客户端 SocketChannel 的 ChannelPipeline 中；

第二步：设置客户端 SocketChannel 的 TCP 参数；

第三步：注册 SocketChannel 到多路复用器。

channelRead 主要执行如上图所示的三个方法，下面我们展开看下 NioSock-

etChannel 的 register 方法，代码如下所示。

```
@Override
@SuppressWarnings("unchecked")
public void channelRead(ChannelHandlerContext ctx, Object msg) {
    Channel child = (Channel) msg;

    child.pipeline().addLast(childHandler);

    for (Entry<ChannelOption<?>, Object> e: childOptions) {
        try {
            if (!child.config().setOption((ChannelOption<Object>) e.getKey(), e.getValue())) {
                logger.warn("Unknown channel option: " + e);
            }
        } catch (Throwable t) {
            logger.warn("Failed to set a channel option: " + child, t);
        }
    }

    for (Entry<AttributeKey<?>, Object> e: childAttrs) {
        child.attr((AttributeKey<Object>) e.getKey()).set(e.getValue());
    }

    child.unsafe().register(child.newPromise());
}

for (Entry<AttributeKey<?>, Object> e: childAttrs) {
    child.attr((AttributeKey<Object>) e.getKey()).set(e.getValue());
}

child.unsafe().register(child.newPromise());
```



The tooltip shows two types: 'Unsafe - io.netty.channel.Channel' and 'AbstractUnsafe - io.netty.channel.AbstractChannel'.

NioSocketChannel 的注册方法与 ServerSocketChannel 的一致，也是将 Channel 注册到 Reactor 线程的多路复用器上，由于注册的操作位是 0，所以，此时 NioSocketChannel 还不能读取客户端发送的消息，那什么时候修改监听操作位为 OP_READ 呢，别着急，继续看代码。

执行完注册操作之后，紧接着会触发 ChannelReadComplete 事件，我们继续分析 ChannelReadComplete 在 ChannelPipeline 中的处理流程：Netty 的 Header 和 Tail 本身不关注 ChannelReadComplete 事件就直接透传，执行完 ChannelReadComplete 后，接着执行 PipeLine 的 read() 方法，最终执行 HeadHandler 的 read() 方法，代码如下：

```
@Override
public void read(ChannelHandlerContext ctx) {
    unsafe.beginRead();
}
```

后面的代码已经在之前的小节已经介绍过，用来修改网络操作位为读操作，创建 `NioSocketChannel` 的时候已经将 `AbstractNioChannel` 的 `readInterestOp` 设置为 `OP_READ`，这样，执行 `selectionKey.interestOps(interestOps | readInterestOp)` 操作时就会把操作位设置为 `OP_READ`。代码如下：

```
/**
 * Create a new instance
 *
 * @param parent      the parent {@link Channel} by which this instance was created. May
 * @param ch          the underlying {@link SelectableChannel} on which it operates
 */
protected AbstractNioByteChannel(Channel parent, EventLoop eventLoop, SelectableChannel ch) {
    super(parent, eventLoop, ch, SelectionKey.OP_READ);
}
```

到此，新接入的客户端连接处理完成，可以进行网络读写等 I/O 操作。

3 Netty 客户端创建

Netty 为了向使用者屏蔽 NIO 通信的底层细节，在和用户交互的边界做了封装，目的就是减少用户开发工作量，降低开发难度。Bootstrap 是 Socket 客户端创建工具类，用户通过 Bootstrap 可以方便的创建 Netty 的客户端并发起异步 TCP 连接操作。

3.1 Netty客户端创建时序图

步骤 1: 用户线程创建 Bootstrap 实例，通过 API 设置创建客户端相关的参数，异步发起客户端连接。

步骤 2: 创建处理客户端连接、I/O 读写的 Reactor 线程组 NioEventLoopGroup，可以通过构造函数指定 I/O 线程的个数，默认为 CPU 内核数的 2 倍；

步骤 3: 通过 Bootstrap 的 ChannelFactory 和用户指定的 Channel 类型创建用于客户端连接的 NioSocketChannel，它的功能类似于 JDK NIO 类库提供的 SocketChannel；

步骤 4: 创建默认的 Channel Handler Pipeline，用于调度和执行网络事件；

步骤 5: 异步发起 TCP 连接, 判断连接是否成功, 如果成功, 则直接将 NioSocketChannel 注册到多路复用器上, 监听读操作位, 用于数据报读取和消息发送; 如果没有立即连接成功, 则注册连接监听位到多路复用器, 等待连接结果;

步骤 6: 注册对应的网络监听状态位到多路复用器;

步骤 7: 由多路复用器在 I/O 现场中轮询各 Channel, 处理连接结果;

步骤 8: 如果连接成功, 设置 Future 结果, 发送连接成功事件, 触发 ChannelPipeline 执行;

步骤 9: 由 ChannelPipeline 调度执行系统和用户的 ChannelHandler, 执行业务逻辑。

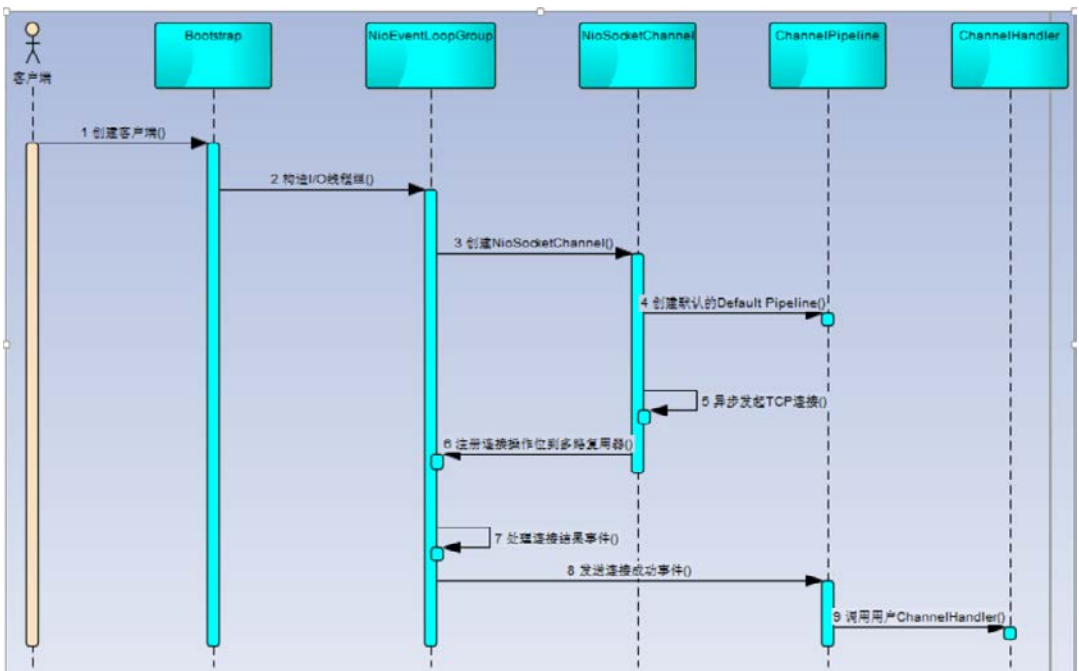


图 3-1 Netty 客户端创建时序图

3.2 Netty客户端创建源码分析

首先, 创建 Bootstrap 的实例, 类似 ServerBootstrap, 客户端也使用 Builder 模式来构造。对于客户端, 由于它不需要监听和处理来自客户端的连接, 所以, 只需要一个 Reactor 线程组即可, 代码如下:

```
// Configure the client.
EventLoopGroup group = new NioEventLoopGroup(1);
Bootstrap b = new Bootstrap();
b.group(group)
```

完成连接辅助类和 Reactor 线程组的初始化操作后，继续设置发起连接的 Channel 为 NioSocketChannel，代码如下：

```
b.group(group)
    .channel(NioSocketChannel.class)
```

如同服务端启动辅助类，客户端辅助类采用工厂模式创建 NioSocketChannel，BootstrapChannelFactory 是 Bootstrap 的内部静态工厂类，用于根据 Channel 的类型和构造函数反射创建新的 NioSocketChannel，代码如下所示：

```
@Override
public T newChannel(EventLoop eventLoop) {
    try {
        Constructor<? extends T> constructor = clazz.getConstructor(EventLoop.class);
        return constructor.newInstance(eventLoop);
    } catch (Throwable t) {
        throw new ChannelException("Unable to create Channel from class " + clazz, t);
    }
}
```

Channel 工厂初始化完成后，设置 TCP 参数，然后设置 Handler，由于此时 NioSocketChannel 还没有真正创建，所以，PipeLine 也没有创建，Netty 预置一个负责创建业务 Handler 的初始化 Handler 工厂到启动辅助类中，当 initChannel 方法被执行时再创建业务 Handler，代码如下：

```
.handler(new ChannelInitializer<SocketChannel>() {
    @Override
    public void initChannel(SocketChannel ch) throws Exception {
        ch.pipeline().addLast(
            //new LoggingHandler(LogLevel.INFO),
            new EchoClientHandler(firstMessageSize));
    }
});
```

一切准备就绪后，发起连接操作，代码如下：

```
private ChannelFuture doConnect(final SocketAddress remoteAddress,
    final SocketAddress localAddress) {
    final ChannelFuture regFuture = initAndRegister();
    final Channel channel = regFuture.channel();
    if (regFuture.cause() != null) {
        return regFuture;
    }

    final ChannelPromise promise = channel.newPromise();
    if (regFuture.isDone()) {
        doConnect0(regFuture, channel, remoteAddress, localAddress,
            promise);
    } else {
        regFuture.addListener(new ChannelFutureListener() {
            @Override
            public void operationComplete(ChannelFuture future)
                throws Exception {
                doConnect0(regFuture, channel, remoteAddress, localAddress,
                    promise);
            }
        });
    }

    return promise;
}
```

第一步，初始化 NioSocketChannel，设置 TCP 参数，注册 SocketChannel 到 Reactor 线程的多路复用器中，代码如下：

```
@Override
Channel createChannel() {
    EventLoop eventLoop = group().next();
    return channelFactory().newChannel(eventLoop);
}
```

初始化 NioSocketChannel，将预置的 Handler 加入到 NioSocketChannel 的 Pipeline 中，设置客户端连接的 TCP 参数，代码如下。

发起注册操作，注册操作在创建服务端的时候已经详细讲解过，这里不再重复讲解。

第二步：判断 NioSocketChannel 是否注册成功，由于是异步注册，通常返回是 False，执行第三步操作，当 NioSocketChannel 注册成功后，发起异步连接操作。

```

void init(Channel channel) throws Exception {
    ChannelPipeline p = channel.pipeline();
    p.addLast(handler());
    final Map<ChannelOption<?>, Object> options = options();
    synchronized (options) {
        for (Entry<ChannelOption<?>, Object> e: options.entrySet()) {
            try {
                if (!channel.config().setOption((ChannelOption<Object>)
                    e.getKey(), e.getValue())) {
                    logger.warn("Unknown channel option: " + e);
                }
            } catch (Throwable t) {
                logger.warn("Failed to set a channel option: " + channel, t);
            }
        }
    }
    final Map<AttributeKey<?>, Object> attrs = attrs();
    synchronized (attrs) {
        for (Entry<AttributeKey<?>, Object> e: attrs.entrySet()) {
            channel.attr((AttributeKey<Object>) e.getKey()).set(e.getValue());
        }
    }
}

channel.eventLoop().execute(new Runnable() {
    @Override
    public void run() {
        if (regFuture.isSuccess()) {
            if (localAddress == null) {
                channel.connect(remoteAddress, promise);
            } else {
                channel.connect(remoteAddress, localAddress, promise);
            }
            promise.addListener(ChannelFutureListener.CLOSE_ON_FAILURE);
        } else {
            promise.setFailure(regFuture.cause());
        }
    }
});
}

```

根据客户端是否指定本地绑定地址执行不同的分支，下面具体分析 AbstractChannel 发起的连接操作，代码如下：

```

public ChannelFuture connect(SocketAddress remoteAddress, SocketAddress localAddress,
    ChannelPromise promise) {
    return pipeline.connect(remoteAddress, localAddress, promise);
}

```

首先调用 NioSocketChannel 的 PipeLine，执行连接操作，最终会调用到 HeadHandler 的 connect 方法，代码如下：

```
public void connect(
    ChannelHandlerContext ctx,
    SocketAddress remoteAddress, SocketAddress localAddress,
    ChannelPromise promise) throws Exception {
    unsafe.connect(remoteAddress, localAddress, promise);
}
```

展开 AbstractNioUnsafe 的 connect 进行分析，代码如下：

```
if (doConnect(remoteAddress, localAddress)) {
    fulfillConnectPromise(promise, wasActive);
} else {
    connectPromise = promise;
    requestedRemoteAddress = remoteAddress;
}
```

首先获取当前的连接状态进行缓存，然后发起连接操作，代码如下：

```
protected boolean doConnect(SocketAddress remoteAddress, SocketAddress localAddress)
    throws Exception {
    if (localAddress != null) {
        javaChannel().socket().bind(localAddress);
    }

    boolean success = false;
    try {
        boolean connected = javaChannel().connect(remoteAddress);
        if (!connected) {
            selectionKey().interestOps(SelectionKey.OP_CONNECT);
        }
        success = true;
        return connected;
    } finally {
        if (!success) {
            doClose();
        }
    }
}
```

大家需要注意的是，SocketChannel 执行 connect() 操作后有三种结果：

- 连接成功，返回True；
- 暂时没有连接上，服务端没有返回ACK应答，连接结果不确定，返回False；

- 连接失败，直接抛出I/O异常。

如果是第二种结果，需要将NioSocketChannel中的selectionKey设置为OP_CONNECT，监听连接结果。

异步连接返回后，需要判断连接结果，如果连接成功，则触发ChannelActive事件，代码如下：

```
private void fulfillConnectPromise(ChannelPromise promise, boolean wasActive) {
    // trySuccess() will return false if a user cancelled the connection attempt.
    boolean promiseSet = promise.trySuccess();
    // Regardless if the connection attempt was cancelled, channelActive() event
    // because what happened is what happened.
    if (!wasActive && isActive()) {
        pipeline().fireChannelActive();
    }
}
```

ChannelActive事件处理在前面章节已经详细说明，最终会将NioSocketChannel中的selectionKey设置为SelectionKey.OP_READ，用于监听网络读操作。

如果没有立即连接上服务端，则执行如下分支：

```
} else {
    connectPromise = promise;
    requestedRemoteAddress = remoteAddress;

    // Schedule connect timeout.
    int connectTimeoutMillis = config().getConnectTimeoutMillis();
    if (connectTimeoutMillis > 0) {
        connectTimeoutFuture = eventLoop().schedule(new Runnable() {
            @Override
            public void run() {
                ChannelPromise connectPromise = AbstractNioChannel.this.connectPromise;
                ConnectTimeoutException cause =
                    new ConnectTimeoutException("connection timed out: " + remoteAddress);
                if (connectPromise != null && connectPromise.tryFailure(cause)) {
                    close(voidPromise());
                }
            }
        }, connectTimeoutMillis, TimeUnit.MILLISECONDS);
    }

    promise.addListener(new ChannelFutureListener() {
```

上面的操作有两个目的：

根据连接超时事件设置定时任务，超时时间到之后触发校验，如果发现连接并没有完成，则关闭连接句柄，释放资源，设置异常堆栈并发起去注册；

设置连接结果监听器，如果接收到连接完成通知则判断连接是否被取消，如果被取消则关闭连接句柄，释放资源，发起取消注册操作。

当服务端返回 ACK 应答后，触发 Selector 轮询出就绪的 SocketChannel，代码如下：

```
if ((readyOps & SelectionKey.OP_CONNECT) != 0) {
    // remove OP_CONNECT as otherwise Selector.select(..)
    // See https://github.com/netty/netty/issues/924
    int ops = k.interestOps();
    ops &= ~SelectionKey.OP_CONNECT;
    k.interestOps(ops);

    unsafe.finishConnect();
}
```

首先将 OP_CONNECT 从 selector 上摘除掉，然后调用 AbstractNioChannel 的 finishConnect 方法，判断异步连接的结果，代码如下：

```
protected void doFinishConnect() throws Exception {
    if (!javaChannel().finishConnect()) {
        throw new Error();
    }
}
```

通过 SocketChannel 的 finishConnect 方法判断连接结果，执行该方法返回三种结果：

- 连接成功返回True；
- 连接失败返回False；
- 发生链路被关闭、链路中断等异常，连接失败。

只要连接失败，就抛出 Error()，由调用方执行句柄关闭等资源释放操作，如果返回成功，则执行 fulfillConnectPromise 方法，该方法之前已经介绍过，

它负责将 SocketChannel 修改为读操作，用来监听网络的读事件，代码如下：

```
private void fulfillConnectPromise(ChannelPromise promise, boolean wasActive) {  
    // trySuccess() will return false if a user cancelled the connection attempt  
    boolean promiseSet = promise.trySuccess();  
    // Regardless if the connection attempt was cancelled, channelActive() event  
    // because what happened is what happened.  
    if (!wasActive && isActive()) {  
        pipeline().fireChannelActive();  
    }  
}
```

如果连接超时时仍然没有接收到服务端的 ACK 应答消息，则由定时任务关闭客户端连接，将 SocketChannel 从 Reactor 线程的多路复用器上摘除，释放资源。

4 Netty 消息的发送和接收

Netty 消息的读取和发送都是非阻塞模式，这是它相比于传统 BIO 最大的优势，下面我们一起分析下 Netty 是如何异步的处理读写操作的。

4.1 异步读取操作

NioEventLoop 作为 Reactor 线程，负责轮询多路复用器，获取就绪的通道执行网络的连接、客户端请求接入、读和写。

当多路复用器检测到读操作后，执行如下方法：不同的 Channel 对应不同的 NioUnsafe：

```
if ((readyOps & (SelectionKey.OP_READ | SelectionKey.OP_ACCEPT)) != 0 || readyOps == 0) {
    unsafe.read();
    if (!ch.isOpen()) {
        // Connection already closed - no need to handle write.
        return;
    }
}
```

此处对应的是 NioByteUnsafe，下面我们进入它的父类 AbstractNioByteChannel 类进行详细分析：

```

public void read() {
    final ChannelConfig config = config();
    final ChannelPipeline pipeline = pipeline();
    final ByteBufAllocator allocator = config.getAllocator();
    final int maxMessagesPerRead = config.getMaxMessagesPerRead();
    RecvByteBufAllocator.Handle allocHandle = this.allocHandle;
    if (allocHandle == null) {
        this.allocHandle = allocHandle = config.getRecvByteBufAllocator().newHandle();
    }
    if (!config.isAutoRead()) {
        removeReadOp();
    }
}

```

首先，获取 NioSocketChannel 的 SocketChannelConfig，它主要用于设置客户端连接的 TCP 参数，接口如下：



我们重点看红框中标出的代码：如果首次调用，从 `SocketChannelConfig` 的 `RecvByteBufAllocator` 中创建 `Handle`。下面我们对 `RecvByteBufAllocator` 做下简单的代码分析：`RecvByteBufAllocator` 默认有两种实现，分别是：`AdaptiveRecvByteBufAllocator` 和 `FixedRecvByteBufAllocator`。由于 `FixedRecvByteBufAllocator` 的实现比较简单，我们重点分析 `AdaptiveRecvByteBufAllocator` 的实现。

顾名思义，AdaptiveRecvByteBufAllocator 指的是缓冲区大小可以动态调整的 ByteBuf 分配器。下面看下它的成员变量：

```
static final int DEFAULT_MINIMUM = 64;
static final int DEFAULT_INITIAL = 1024;
static final int DEFAULT_MAXIMUM = 65536;

private static final int INDEX_INCREMENT = 4;
private static final int INDEX_DECREMENT = 1;
```

它分别定义了三个系统默认值：最小缓冲区长度 64 字节、初始容量 1024 字节、最大容量 65536 字节。

还定义了两个动态调整容量时的步进参数：扩张的步进索引为 4、收缩的步进索引为 1。

最后，定义了长度的向量表 SIZE_TABLE 并初始化它，它的初始值如下：

```
0-->16 1-->32 2-->48 3-->64 4-->80 5-->96 6-->112 7-->128 8-->144
9-->160
10-->176 11-->192 12-->208 13-->224 14-->240 15-->256 16-->272 17-->288 18-->304
19-->320 20-->336 21-->352 22-->368 23-->384 24-->400 25-->416 26-->432 27-->448
28-->464 29-->480 30-->496 31-->512 32-->1024 33-->2048 34-->4096
35-->8192 36-->16384
37-->32768 38-->65536 39-->131072 40-->262144 41-->524288 42-->1048576
43-->2097152 44-->4194304 45-->8388608
46-->16777216 47-->33554432 48-->67108864 49-->134217728 50-->268435456
51-->536870912 52-->1073741824
```

向量数组的每个值都对应一个 Buffer 容量，当容量小于 512 的时候，由于缓冲区已经比较小，需要降低步进值，容量每次下调的幅度要小些；当大于 512 时，说明需要解码的消息码流比较大，这时采用调大步进幅度的方式减少动态扩张的

频率，所以它采用 512 的倍数进行扩张。

接下来我们重点分析下 AdaptiveRecvByteBufAllocator 的方法：方法一：
getSizeTableIndex(final int size)，代码如下：

```
private static int getSizeTableIndex(final int size) {
    for (int low = 0, high = SIZE_TABLE.length - 1;;) {
        if (high < low) {
            return low;
        }
        if (high == low) {
            return high;
        }

        int mid = low + high >>> 1;
        int a = SIZE_TABLE[mid];
        int b = SIZE_TABLE[mid + 1];
        if (size > b) {
            low = mid + 1;
        } else if (size < a) {
            high = mid - 1;
        } else if (size == a) {
            return mid;
        } else {
            return mid + 1;
        }
    }
}
```

根据容量 Size 查找容量向量表对应的索引：这是个典型的二分查找法，由于它的算法非常经典，也比较简单，此处不再赘述。

下面我们分析下它的内部静态类 HandleImpl，首先，看下它的成员变量：

```
private static final class HandleImpl implements Handle {
    private final int minIndex;
    private final int maxIndex;
    private int index;
    private int nextReceiveBufferSize;
    private boolean decreaseNow;
}
```

它有五个成员变量，分别是：对应向量表的最小索引、最大索引、当前索引、下一次预分配的 Buffer 大小和是否立即执行容量收缩操作。

我们重点分析它的 record(int actualReadBytes) 方法：当 NioSocketChannel 执行完读操作后，会计算获得本次轮询读取的总字节数，

它就是参数 `actualReadBytes`，执行 `record` 方法，根据实际读取的字节数对 `ByteBuf` 进行动态伸缩和扩张，代码如下：

```
public void record(int actualReadBytes) {
    if (actualReadBytes <= SIZE_TABLE[Math.max(0, index
        - INDEX_DECREMENT - 1)]) {
        if (decreaseNow) {
            index = Math.max(index - INDEX_DECREMENT, minIndex);
            nextReceiveBufferSize = SIZE_TABLE[index];
            decreaseNow = false;
        } else {
            decreaseNow = true;
        }
    } else if (actualReadBytes >= nextReceiveBufferSize) {
        index = Math.min(index + INDEX_INCREMENT, maxIndex);
        nextReceiveBufferSize = SIZE_TABLE[index];
        decreaseNow = false;
    }
}
```

首先，对当前索引做步进缩减，然后获取收缩后索引对应的容量，与实际读取的字节数进行比对，如果发现小于收缩后的容量，则重新对当前索引进行赋值，取收缩后的索引和最小索引中的较大者作为最新的索引，然后，为下一次缓冲区容量分配赋值——新的索引对用容量向量表中的容量。相反，如果当前实际读取的字节数大于之前预分配的初始容量，则说明实际分配的容量不足，需要动态扩张。重新计算索引，选取当前索引 + 扩张步进 和 最大索引中的较小作为当前索引值，然后对下次缓冲区的容量值进行重新分配，完成缓冲区容量的动态扩张。

通过上述分析我们得知，`AdaptiveRecvByteBufAllocator` 就是根据本次读取的实际字节数对下次接收缓冲区的容量进行动态分配。

使用动态缓冲区分配器的优点如下：

1. Netty作为一个通用的NIO框架，并不对客户的应用场景进行假设，你可能使用它做流媒体传输，也可能用它做聊天工具，不同的应用场景，传输的码流大小千差万别；无论初始化分配的是32K还是1M，都会随着应用场景的变化而变得不适应，因此，Netty根据上次实际读取的码流大小对下次的接收Buffer缓冲区进行预测和调整，能够最大限度的满足不同行业的应

用场景；

2. 性能更高，容量过大会导致内存占用开销增加，后续的Buffer处理性能会下降；容量过小时需要频繁的内存扩张来接收大的请求消息，同样会导致性能下降；
3. 更节约内存：设想，假如通常情况下请求消息平均值为1M左右，接收缓冲区大小为1.2M；突然某个客户发送了一个10M的流媒体附件，接收缓冲区扩张为10M以接纳该附件，如果缓冲区不能收缩，每次缓冲区创建都会分配10M的内存，但是后续所有的消息都是1M左右，这样会导致内存的浪费，如果并发客户端过多、Reactor线程个数过多，可能会发生内存溢出，最终宕机。

分析完 AdaptiveRecvByteBufAllocator，我们继续分析读操作：

```
try {  
    int byteBufCapacity = allocHandle.guess();  
    int totalReadAmount = 0;  
    do {  
        byteBuf = allocator.ioBuffer(byteBufCapacity);
```

首先通过接收缓冲区分配器的 Handler 计算获得下次预分配的缓冲区容量 byteBufCapacity，紧接着根据缓冲区容量进行缓冲区分配，Netty 的缓冲区种类很多，此处重点介绍的是消息的读取，因此对缓冲区不展开说明。

接收缓冲区 ByteBuf 分配完成后，进行消息的异步读取，代码如下：

```
int localReadAmount = doReadBytes(byteBuf);
```

它是个抽象方法，具体实现在 NioSocketChannel 中，代码如下：

```
protected int doReadBytes(ByteBuf byteBuf) throws Exception {  
    return byteBuf.writeBytes(javaChannel(), byteBuf.writableBytes());  
}
```

其中 javaChannel() 返回的是 SocketChannel，byteBuf.writableBytes() 返回本次可读的最大长度，我们继续展开看最终是如何从 Channel 中读取码流的，代码如下：

```

public int writeBytes(ScatteringByteChannel in, int length)
    throws IOException {
    ensureWritable(length);
    int writtenBytes = setBytes(writerIndex, in, length);
    if (writtenBytes > 0) {
        writerIndex += writtenBytes;
    }
    return writtenBytes;
}

```

对 setBytes 方法展开代码如下：

```

public int setBytes(int index, ScatteringByteChannel in, int length)
    throws IOException {
    ensureAccessible();
    try {
        return in.read((ByteBuffer) internalNioBuffer().clear().
            position(index).limit(index + length));
    } catch (ClosedChannelException e) {
        return -1;
    }
}

```

由于 SocketChannel 的 read 方法参数是 JAVA NIO 的 ByteBuffer，所以，需要先将 Netty 的 ByteBuf 转换成 JDK 的 ByteBuffer，随后调用 ByteBuffer 的 clear 方法对指针进行重置用于新消息的读取，随后将 position 指针指到初始读的 index，读取的上限设置为 index + 读取的长度。最后调用 read 方法将 SocketChannel 中就绪的码流读取到 ByteBuffer 中，完成消息的读取，返回读取的字节数。

完成消息的异步读取后，需要对本次读取的字节数进行判断，有三种可能：

1. 返回0，表示没有就绪的消息可读；
2. 返回值大于0，读到了消息；
3. 返回值-1，表示发生了IO异常，读取失败。

下面我们继续看 Netty 的处理逻辑，首先对读取的字节数进行判断，如果等于或者小于 0，表示没有就绪的消息可读或者发生了 IO 异常，此时需要释放接收缓冲区，如果读取的字节数小于 0，则需要将 close 状态位置位，用于关闭连接，释放句柄资源。置位完成之后，退出循环：

```

if (localReadAmount <= 0) {
    // not was read release the buffer
    byteBuf.release();
    close = localReadAmount < 0;
    break;
}

```

完成一次异步读之后，就会触发一次 ChannelRead 事件，这里特别需要提醒大家的是，完成一次读操作，并不意味着读到了一条完整的消息，因为 TCP 底层存在组包和粘包，所以，一次读操作可能包含多条消息，也可能是一条不完整的消息，所以，不要把它跟读取的消息个数等同起来。我曾经发现有同事在没有做任何半包处理的情况下，以 ChannelRead 的触发次数做计数器来进行性能分析和统计，是完全错误的。当然，如果你使用了针对半包的 Decode 类或者自己做了特殊封装，对 ChannelRead 事件进行拦截，屏蔽 Netty 的默认机制，也能够实现一次 ChannelRead 对应一条完整消息的效果，此处也不再展开说明了，当你掌握了 Netty 的编解码技巧之后，自然就知道如何实现这种效果了。触发和完成 ChannelRead 事件调用之后，将接收缓冲区释放。

因为一次读操作未必能够完成 TCP 缓冲区的全部读取工作，所以，读操作在循环体中进行，每次读取操作完成之后，会对读取的字节数进行累加，代码如下：

```

if (totalReadAmount >= Integer.MAX_VALUE - localReadAmount) {
    // Avoid overflow.
    totalReadAmount = Integer.MAX_VALUE;
    break;
}

totalReadAmount += localReadAmount;
if (localReadAmount < writable) {
    // Read less than what the buffer can hold,
    // which might mean we drained the recv buffer completely.
    break;
}

```

在累加之前，需要对长度上限做保护，如果累计读取的字节数已经发生溢出，则将读取到的字节数设置为整形的最大值，然后退出循环，原因是本次循环已经读取过多的字节，需要退出。否则会影响后面排队的 Task 任务和写操作的执行。如果没有溢出，则执行累加操作。

最后，对本次读取的字节数进行判断，如果小于缓冲区可写的容量，说明

TCP 缓冲区已经没有就绪的字节可读，读取操作已经完成，需要退出循环。如果仍然有未读的消息，则继续执行读操作。连续的读操作会阻塞排在后面的任务队列中待执行的 Task，以及写操作，所以，对连续读操作做了上限控制，默认值为 16 次，无论 TCP 缓冲区有多少码流需要读取，只要连续 16 次没有读完，都需要强制退出，等待下次 selector 轮询周期再执行。

```
} while (++ messages < maxMessagesPerRead);
```

完成多路复用器本轮读操作之后，触发 ChannelReadComplete 事件。随后调用接收缓冲区容量分配器的 Handler 的记录方法，将本次读取的总字节数传入到 record() 方法中进行缓冲区的动态分配，为下一次读取选取更加合适的缓冲区容量，代码如下：

```
allocHandle.record(totalReadAmount);
```

上面我们提到，如果读到的返回值为 -1，表明发生了 I/O 异常，需要关闭连接，释放资源，代码如下：

```
if (close) {  
    closeOnRead(pipeline);  
    close = false;  
}
```

4.2 异步消息发送

Netty 的写操作和将消息真正刷新到 SocketChannel 中是分开的，因此我们分成两个小结来介绍，首先介绍消息的写操作。

4.2.1 异步消息发送

下面我们从 ChannelHandlerContext 开始分析，首先调用它的 write 方法，异步发送消息，代码如下：

```
public ChannelFuture write(Object msg, ChannelPromise promise) {  
    DefaultChannelHandlerContext next = findContextOutbound(MASK_WRITE);  
    next.invoker.invokeWrite(next, msg, promise);  
    return promise;  
}
```

类似 Mina 的 FilterChain，它实际上是个职责链，消息在职责链中传递，最终它会调用 HeadHandler 的 write 方法，代码如下：

```
public void write(ChannelHandlerContext ctx, Object msg,
    ChannelPromise promise) throws Exception {
    unsafe.write(msg, promise);
}
```

它由子类 AbstractUnsafe 实现，代码如下：

```
public void write(Object msg, ChannelPromise promise) {
    if (!isActive()) {
        // Mark the write request as failure if the channel
        if (isOpen()) {
            promise.tryFailure(NOT_YET_CONNECTED_EXCEPTION);
        } else {
            promise.tryFailure(CLOSED_CHANNEL_EXCEPTION);
        }
        // release message now to prevent resource-leak
        ReferenceCountUtil.release(msg);
    } else {
        outboundBuffer.addMessage(msg, promise);
    }
}
```

首先对链路的状态进行判断，如果已经断开连接，则需要设置回调结果异常信息，同时，释放需要发送的消息。注意：此处的消息通常是经过编码后的 ByteBuf，因此，需要释放。

如果链路正常，则将需要发送的 ByteBuf 加入到 outboundBuffer 中，下面，我们重点分析 ChannelOutboundBuffer 的 addMessage 方法。代码如下：

```
void addMessage(Object msg, ChannelPromise promise) {
    int size = channel.estimatorHandle().size(msg);
    if (size < 0) {
        size = 0;
    }
    Entry e = buffer[tail++];
    e.msg = msg;
    e.pendingSize = size;
    e.promise = promise;
    e.total = total(msg);
    tail &= buffer.length - 1;
    if (tail == flushed) {
        addCapacity();
    }
    // increment pending bytes after adding message to
    // See https://github.com/netty/netty/issues/1619
    incrementPendingOutboundBytes(size);
}
```


首先，我们获取 ByteBuf 的可读字节数，实际上也就是需要发送的字节数。

然后，从环形 Entry 数组中获取可用的 Entry，将指针 +1，接着进行一系列的赋值操作，例如将 Entry 的 Message 设置为需要发送的 ByteBuf 等。设置完成后需要进行一次判断，如果当前指针已经达到唤醒数组的尾部，即：tail = buffer.length；此时需要重新将指针调整为起始位置 0。由于环形数组的初始容量为 32，后面容量的扩张是 32 的 N 倍，所以通过 & 操作就能将指针重新指到起始位置，实现环形队列，代码如下：

```
tail &= buffer.length - 1;
```

指针重绕后，需要对尾部指针 tail 和需要刷新的位置 flushed 进行判断，如果两者相等，说明指针重绕后已经到达需要刷新的位置，再继续使用就会覆盖尚未发送的消息，因此，需要对环形队列进行动态扩容，动态扩展的代码如下：

```
private void addCapacity() {
    int p = flushed;
    int n = buffer.length;
    int r = n - p; // number of elements to the right of p
    int s = size();

    int newCapacity = n << 1;
    if (newCapacity < 0) {
        throw new IllegalStateException();
    }

    Entry[] e = new Entry[newCapacity];
    System.arraycopy(buffer, p, e, 0, r);
    System.arraycopy(buffer, 0, e, r, p);
    for (int i = n; i < e.length; i++) {
        e[i] = new Entry();
    }

    buffer = e;
    flushed = 0;
    unflushed = s;
    tail = n;
}
```

首先，保存需要刷新的位置索引，计算还有多少个消息没有被刷新，然后执行扩容操作，将环形数组的 Size 扩展为原来的 2 倍。扩容以后，需要对新的环形数组进行填充，填充分为三步：

1. 将尚未刷新的消息拷贝到数组的首部；

2. 原来数组中已经刷新并释放的Entry可以重用，所以，将其拷贝到尚未刷新消息的后面；
3. 最后扩容的数组全部重新初始化。

对扩容后的数组初始化后，需要对指针进行重新置位，具体如下：

由于尚未刷新的消息在数组首部，所以 flushed 为 0；

由于未刷新的消息从 0 开始，所以 $\text{unflushed} = \text{unflushed} - \text{flushed} \& \text{buffer.length} - 1$ ；

下次新的消息写入需要放入扩容后的数组中，所以 $\text{tail} = \text{buffer.length}$

将需要发送的消息写入环形发送数组之后，计算当前需要发送消息的总字节数是否达到一次发送的高水位线，如果达到，触发 `channelWritabilityChanged` 事件，代码如下：

```
void incrementPendingOutboundBytes(int size) {
    // Cache the channel and check for null to make sure we not
    // produce a NPE in case of the Channel gets
    // recycled while process this method.
    Channel channel = this.channel;
    if (size == 0 || channel == null) {
        return;
    }
    long oldValue = totalPendingSize;
    long newWriteBufferSize = oldValue + size;
    while (!TOTAL_PENDING_SIZE_UPDATER.compareAndSet(this, oldValue,
        newWriteBufferSize)) {
        oldValue = totalPendingSize;
        newWriteBufferSize = oldValue + size;
    }
    int highWaterMark = channel.config().getWriteBufferHighWaterMark(

    if (newWriteBufferSize > highWaterMark) {
        if (WRITABLE_UPDATER.compareAndSet(this, 1, 0)) {
            channel.pipeline().fireChannelWritabilityChanged();
        }
    }
}
```

这段代码理解起来非常简单，不再展开说明。只对红框中标出的部分做解释：

它仿照了 JDK1.5 以后新增的原子类的自旋操作解决多线程并发操作问题，循环判断，如果需要更新的变量值没有发生变化并且更新成功退出，否则取其它线程更新后的新值重新计算并重新赋值，这个就是自旋，通过它可以解决多线程

并发修改一个变量的无锁化问题。

至此，我们完成了消息异步发送的代码分析，接下来，我们继续分析消息的刷新操作，flush 负责将发送环形数组中缓存的消息写入到 SocketChannel 中发送给对方。

4.2.2 Flush 操作

Flush 操作负责将 ByteBuffer 消息写入到 SocketChannel 中发送给对方，下面我们首先从发起 Flush 操作的类入口，进行详细分析。

DefaultChannelHandlerContext 的 flush 方法，最终会调用的 HeadHandler 的 flush 操作，代码如下：

```
public void flush(ChannelHandlerContext ctx) throws Exception {
    unsafe.flush();
}
```

重点分析 AbstractUnsafe 的 flush 操作，代码如下：

```
public void flush() {
    ChannelOutboundBuffer outboundBuffer = this.outboundBuffer;
    if (outboundBuffer == null) {
        return;
    }

    outboundBuffer.addFlush();
    flush0();
}
```

首先将发送环形数组的 unflushed 指针修改为 tail，标识本次要发送的消息范围。然后调用 flush0 进行发送，由于 flush0 代码非常简单，我们重点分析 doWrite 方法，代码如下：

```
protected void doWrite(ChannelOutboundBuffer in) throws Exception {
    for (;;) {
        // Do non-gathering write for a single buffer case.
        final int msgCount = in.size();
        if (msgCount <= 1) {
            super.doWrite(in);
            return;
        }
    }
}
```

首先计算需要发送的消息个数 (unflushed - flush)，如果只有 1 个消息需要发送，则调用父类的写操作，我们分析 AbstractNioByteChannel 的 doWrite() 方法，代码如下：

```
for (;;) {
    Object msg = in.current(true);
    if (msg == null) {
        // Wrote all messages.
        clearOpWrite();
        break;
    }
}
```

因为只有一条消息需要发送，所以直接从 ChannelOutboundBuffer 中获取当前需要发送的消息，代码如下：

```
public Object current(boolean preferDirect) {
    if (isEmpty()) {
        return null;
    } else {
        // TODO: Think of a smart way to handle ByteBuffer messages
        Object msg = buffer[flushed].msg;
        if (threadLocalDirectBufferSize <= 0 || !preferDirect) {
            return msg;
        }
        if (msg instanceof ByteBuffer) {
            ByteBuffer buf = (ByteBuffer) msg;
            if (buf.isDirect()) {
                return buf;
            } else {
                int readableBytes = buf.readableBytes();
                if (readableBytes == 0) {
                    return buf;
                }
            }
        }
    }
}
```

首先，获取需要发送的消息，如果消息为 ByteBuffer 且它分配的是 JDK 的非堆内存，则直接返回。

对返回的消息进行判断，如果为空，说明该消息已经发送完成并被回收，然后执行清空 OP_WRITE 操作位的 clearOpWrite 方法，代码如下：

```
protected final void clearOpWrite() {
    final SelectionKey key = selectionKey();
    final int interestOps = key.interestOps();
    if ((interestOps & SelectionKey.OP_WRITE) != 0) {
        key.interestOps(interestOps & ~SelectionKey.OP_WRITE);
    }
}
```

继续向下分析，如果需要发送的 ByteBuf 已经没有可写的字节，说明已经发送完成，将该消息从环形队列中删除，然后继续循环。下面我们分析下 ChannelOutboundBuffer 的 remove 方法：

```
public boolean remove() {
    if (isEmpty()) {
        return false;
    }
    Entry e = buffer[flushed];
    Object msg = e.msg;
    if (msg == null) {
        return false;
    }
    ChannelPromise promise = e.promise;
    int size = e.pendingSize;
    e.clear();
    flushed = flushed + 1 & buffer.length - 1;
    safeRelease(msg);
    promise.trySuccess();
    decrementPendingOutboundBytes(size);
    return true;
}
```

首先判断环形队列中是否还有需要发送的消息，如果没有，则直接返回。如果非空，则首先获取 Entry，然后对其进行资源释放，同时把需要发送的索引 flushed 进行更新。所有操作执行完之后，调用 decrementPendingOutboundBytes 减去已经发送的字节数，该方法跟 incrementPendingOutboundBytes 类似，会进行发送低水位的判断和事件通知，此处不再赘述。

我们接着继续对消息的发送进行分析，代码如下：首先将半包标致设置为 false：

```
boolean setOpWrite = false;
boolean done = false;
long flushedAmount = 0;
if (writeSpinCount == -1) {
    writeSpinCount = config().getWriteSpinCount();
}
for (int i = writeSpinCount - 1; i >= 0; i --) {
    int localFlushedAmount = doWriteBytes(buf);
    if (localFlushedAmount == 0) {
        setOpWrite = true;
        break;
    }
    flushedAmount += localFlushedAmount;
    if (!buf.isReadable()) {
        done = true;
        break;
    }
}
```

从 DefaultSocketChannelConfig 中获取循环发送的次数，进行循环发送，对发送方法 doWriteBytes 展开分析，如下：

```
protected int doWriteBytes(ByteBuf buf) throws Exception {  
    final int expectedWrittenBytes = buf.readableBytes();  
    final int writtenBytes = buf.readBytes(javaChannel(), expectedWrittenBytes);  
    return writtenBytes;  
}
```

对于红框中的代码说明如下：ByteBuf 的 readBytes() 方法的功能是将当前 ByteBuf 中的可写字节数组写入到指定的 Channel 中。方法的第一个参数是 Channel，此处就是 SocketChannel，第二个参数是写入的字节数组长度，它等于 ByteBuf 的可读字节数，返回值是写入的字节个数。由于我们将 SocketChannel 设置为异步非阻塞模式，所以写操作不会阻塞。

从写操作中返回，需要对写入的字节数进行判断，如果为 0，说明 TCP 发送缓冲区已满，不能继续再向里面写入消息，因此，将写半包标致设置为 true，然后退出循环，执行后续排队的其它任何或者读操作，等待下一次 Selector 的轮询继续触发写操作。

对写入的字节数进行累加，判断当前的 ByteBuf 中是否还有没有发送的字节，如果没有可发送的字节，则将 done 设置为 true，退出循环。

从循环发送状态退出后，首先根据实际发送的字节数更新发送进度，实际就是发送的字节数和需要发送的字节数的一个比值。执行完成进度更新后，判断本轮循环是否将需要发送的消息中所有需要发送的字节全部发送完成，如果发送完成，则将该消息从循环队列中删除；否则，将设置多路复用器的 OP_WRITE 操作位，用于通知 Reactor 线程还有没有发送完成的消息，需要继续发送，直到全部发送完成。

好，到此我们分析完了单条消息的发送，现在我们重新将注意力转回到 NioSocketChannel，看看多条消息的发送过程，代码如下：

```
ByteBuffer[] nioBuffers = in.nioBuffers();  
if (nioBuffers == null) {  
    super.doWrite(in);  
    return;  
}
```


从 ChannelOutboundBuffer 获取需要发送的 ByteBuffer 列表，由于 Netty 使用的是 ByteBuf，因此，需要做下内部类型转换，代码如下：

```
public ByteBuffer[] nioBuffers() {
    long nioBufferSize = 0;
    int nioBufferCount = 0;
    final int mask = buffer.length - 1;
    final ByteBufAllocator alloc = channel.alloc();
    ByteBuffer[] nioBuffers = this.nioBuffers;
    Object m;
    int i = flushed;
    while (i != unflushed && (m = buffer[i].msg) != null) {
        if (!(m instanceof ByteBuf)) {
            this.nioBufferCount = 0;
            this.nioBufferSize = 0;
            return null;
        }
    }
```

声明各种局部变量并赋值，从 flushed 开始循环获取需要发送的 ByteBuf。首先对要发送的 Message 进行判断，如果不是 Netty 的 ByteBuf，则返回空。

```
Entry entry = buffer[i];
ByteBuf buf = (ByteBuf) m;
final int readerIndex = buf.readerIndex();
final int readableBytes = buf.writerIndex() - readerIndex;

if (readableBytes > 0) {
    nioBufferSize += readableBytes;
    int count = entry.count;
    if (count == -1) {
        entry.count = count = buf.nioBufferCount();
    }
}
```

获取可写的字节个数，如果大于 0，对需要发送的缓冲区字节总数进行累加。然后从当前 Entry 中获取 ByteBuf 包含的最大 ByteBuffer 个数。

对包含的 ByteBuffer 个数进行累加，如果超过 ChannelOutboundBuffer 预先分配的数组上限，则进行数组扩张。扩张的代码如下：

```

private static ByteBuffer[] expandNioBufferArray(ByteBuffer[] array,
    int neededSpace, int size) {
    int newCapacity = array.length;
    do {
        // double capacity until it is big enough
        // See https://github.com/netty/netty/issues/1890
        newCapacity <= 1;
        if (newCapacity < 0) {
            throw new IllegalStateException();
        }
    } while (neededSpace > newCapacity);
    ByteBuffer[] newArray = new ByteBuffer[newCapacity];
    System.arraycopy(array, 0, newArray, 0, size);
    return newArray;
}

```

由于频繁的数组扩张会导致频繁的数组拷贝，影响性能，所以，Netty 采用了翻倍扩张的方式，新的数组创建之后，将老的数据内容拷贝到新创建的数组中返回。

ByteBuffer 创建完成之后，需要将要刷新的 ByteBuf 转换成 ByteBuffer 并存到发送数据中。由于 ByteBuf 的实现不同，所以，它们内部包含的 ByteBuffer 个数是不同的，例如 UnpooledHeapByteBuf，它基于 JVM 堆内存的字节数组实现，只包含 1 个 ByteBuffer。对 Entry 中缓存的 ByteBuffer 进行判断，如果为空，则调用 ByteBuf 的 internalNioBuffer 方法，将当前的 ByteBuf 转换为 JDK 的 ByteBuffer，我们以 UnpooledHeapByteBuf 为例看下 internalNioBuffer 的实现：

```

public ByteBuffer internalNioBuffer(int index, int length) {
    return (ByteBuffer) internalNioBuffer().clear().position(index).limit(index + length);
}

```

获取 ByteBuffer 实例，然后调用它的 clear() 方法对它的指针进行初始化，随后将 Position 指针设置为 index，limit 指针设置为 index + length。这些初始化操作完成之后 ByteBuffer 就可以被正确的读写。

下面我们看另一个分支，如果 ByteBuf 包含 NIO ByteBuffer 数组，那就获取 Entry 缓存的 ByteBuffer 数组，如果为空，则从当前需要刷新的 ByteBuf 中

获取它的 `ByteBuffer` 数组。完成赋值操作后，调用 `fillBufferArray` 进行赋值。

对循环变量 `i` 赋值，完成本轮循环，代码如下：

```
i = i + 1 & mask;
```

当 `i = unflushed` 时，说明需要刷新的消息全部赋值完成，循环执行结束。

对 `ByteBuffer` 数组进行判断，看是否还有单个需要发送的消息，如果没有则直接返回，有则发送：

```
ByteBuffer[] nioBuffers = in.nioBuffers();  
if (nioBuffers == null) {  
    super.doWrite(in);  
    return;  
}
```

在批量发送缓冲区的消息之前，先对一系列的局部变量进行赋值，首先，获取需要发送的 `ByteBuffer` 数组个数 `nioBufferCnt`，然后，从 `ChannelOutboundBuffer` 中获取需要发送的总字节数，从 `NioSocketChannel` 中获取 `NIO` 的 `SocketChannel`，是否发送完成标识设置为 `false`，是否有写半包标识设置为 `false`。

```
int nioBufferCnt = in.nioBufferCount();  
long expectedWrittenBytes = in.nioBufferSize();  
  
final SocketChannel ch = javaChannel();  
long writtenBytes = 0;  
boolean done = false;  
boolean setOpWrite = false;
```

继续分析循环发送的代码，代码如下：

```
for (int i = config().getWriteSpinCount() - 1; i >= 0; i --) {  
    final long localWrittenBytes = ch.write(nioBuffers, 0, nioBufferCnt);
```

就像循环读一样，我们需要对一次 `Selector` 轮询的写操作次数进行上限控制，因为如果 `TCP` 的发送缓冲区满，`TCP` 处于 `KEEP-ALIVE` 状态，消息是发送不出去的，如果不对上限进行控制，就会常时间的处于发送状态，`Reactor` 线程无法及时读取其它消息和执行排队的 `Task`。所以，我们必须对循环次数上限做控制。

调用 NIO SocketChannel 的 write 方法，它有三个参数：第一个是需要发送的 ByteBuffer 数组，第二个是数组的偏移量，第三个参数是发送的 ByteBuffer 个数。返回值是写入 SocketChannel 的字节个数。

下面对写入的字节进行判断，如果为 0，说明 TCP 发送缓冲区已满，再写很有可能还是写不进去，因此从循环中跳出，同时将写半包标识设置为 True，用于向多路复用器注册写操作位，告诉多路复用器有没发完的半包消息，你要继续轮询出就绪的 SocketChannel 继续发送：

```
final long localWrittenBytes = ch.write(nioBuffers, 0, nioBufferCnt);
if (localWrittenBytes == 0) {
    setOpWrite = true;
    break;
}
```

发送操作完成后进行两个计算：需要发送的字节数要减去已经发送的字节数，发送的字节总数 + 已经发送的字节数。更新完这两个变量后，判断缓冲区中所有的消息是否已经发送完成，如果是，则把发送完成标识设置为 True 同时退出循环。如果没有发送完成，则继续循环。

从循环发送中退出之后，首先对发送完成标识 done 进行判断，如果发送完成，则循环释放已经发送的消息，代码如红框中标识所示：

环形数组的发送缓冲区释放完成后，取消半包标识，告诉多路复用器消息已经全部发送完成。

```
if (done) {
    // Release all buffers
    for (int i = msgCount; i > 0; i --) {
        in.remove();
    }

    // Finish the write loop if no new messages were
    if (in.isEmpty()) {
        clearOpWrite();
        break;
    }
}
```

当缓冲区中的消息没有发送完成，甚至某个 ByteBuffer 只发送了一半，出现了半包发送，该怎么办？下面我们继续看看 Netty 是如何处理的。

```

for (int i = msgCount; i > 0; i --) {
    final ByteBuf buf = (ByteBuf) in.current();
    final int readerIndex = buf.readerIndex();
    final int readableBytes = buf.writerIndex() - readerIndex;

    if (readableBytes < writtenBytes) {
        in.progress(readableBytes);
        in.remove();
        writtenBytes -= readableBytes;
    } else if (readableBytes > writtenBytes) {
        buf.readerIndex(readerIndex + (int) writtenBytes);
        in.progress(writtenBytes);
        break;
    } else { // readableBytes == writtenBytes
        in.progress(readableBytes);
        in.remove();
        break;
    }
}

```

首先，我们循环遍历发送缓冲区，对消息的发送结果进行分析，下面具体展开进行说明：

1. 从ChannelOutboundBuffer弹出第一条发送的ByteBuf；然后获取该ByteBuf的可读索引和可读字节数；
2. 对可读字节数和发送的总字节数进行判断，如果发送的字节数大于可读的字节数，说明它已经被完全发送出去，更新ChannelOutboundBuffer的发送进度信息，将已经发送的ByteBuf删除，释放相关资源，最后，发送的字节数要减去第一条发送的字节数，就是后面消息发送的总字节数；然后继续循环判断第二条消息、第三条消息.....
3. 如果可读的消息大于已经发送的总消息数，说明这条消息没有被完全发送成功，也就是出现了所谓的“写半包”，此时，需要更新可读的索引为当前索引 + 已经发送的总字节数，然后更新ChannelOutboundBuffer的进度信息，退出循环；
4. 如果可读字节数等于已经发送的字节数总和，则说明最后一次发送的消息是个全包消息，更新发送进度信息，将最后一条完全发送的消息从缓冲区

中删除，最后退出循环。

最后，因为缓冲区中待刷新的消息没有全部发送完成，所以需要更新 SocketChannel 的注册监听位，将其修改为 OP_WRITE，在下一次轮询中继续发送没有发送出去的消息。

5 Netty 线程模型

当我们讨论 Netty 线程模型的时候，一般首先会想到的是经典的 Reactor 线程模型，尽管不同的 NIO 框架对于 Reactor 模式的实现存在差异，但本质上还是遵循了 Reactor 的基础线程模型。

下面让我们一起回顾经典的 Reactor 线程模型。

5.1 Reactor 线程模型

5.1.1 Reactor 单线程模型

Reactor 单线程模型，是指所有的 I/O 操作都在同一个 NIO 线程上面完成。NIO 线程的职责如下。

- 作为NIO服务端，接收客户端的TCP连接；
- 作为NIO客户端，向服务端发起TCP连接；
- 读取通信对端的请求或者应答消息；
- 向通信对端发送消息请求或者应答消息。

Reactor 单线程模型如图 5-1 所示。

由于 Reactor 模式使用的是异步非阻塞 I/O，所有的 I/O 操作都不会导致阻

塞，理论上一个线程可以独立处理所有 I/O 相关的操作。从架构层面看，一个 NIO 线程确实可以完成其承担的职责。例如，通过 Acceptor 类接收客户端的 TCP 连接请求消息，当链路建立成功之后，通过 Dispatch 将对应的 ByteBuffer 派发到指定的 Handler 上，进行消息解码。用户线程消息编码后通过 NIO 线程将消息发送给客户端。

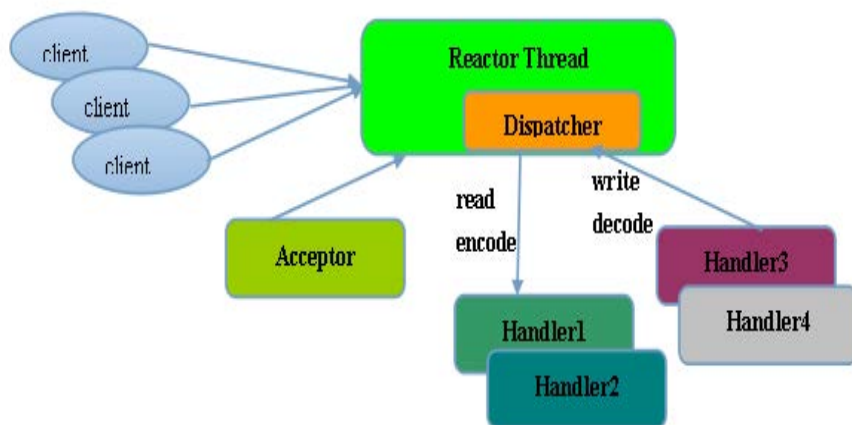


图 5-1 Reactor 单线程模型

在一些小容量应用场景下，可以使用单线程模型。但是这对于高负载、大并发的应用场景却不合适，主要原因如下：

- 一个NIO线程同时处理成百上千的链路，性能上无法支撑，即便NIO线程的CPU负荷达到100%，也无法满足海量消息的编码、解码、读取和发送。
- 当NIO线程负载过重之后，处理速度将变慢，这会导致大量客户端连接超时，超时之后往往会进行重发，这更加重了NIO线程的负载，最终会导致大量消息积压和处理超时，成为系统的性能瓶颈。
- 可靠性问题：一旦NIO线程意外跑飞，或者进入死循环，会导致整个系统通信模块不可用，不能接收和处理外部消息，造成节点故障。

为了解决这些问题，演进出了 Reactor 多线程模型。下面我们一起学习下 Reactor 多线程模型。

5.1.2 Rector 多线程模型

Rector 多线程模型与单线程模型最大的区别就是有一组 NIO 线程来处理 I/O

操作，它的原理如图 5-2 所示。

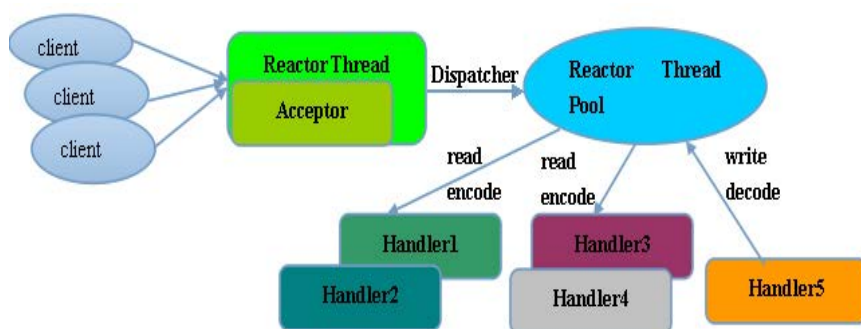


图 5-2 Reactor 多线程模型

Reactor 多线程模型的特点如下。

- 有专门一个NIO线程——Acceptor线程用于监听服务端，接收客户端的TCP连接请求。
- 网络I/O操作——读、写等由一个NIO线程池负责，线程池可以采用标准的JDK线程池实现，它包含一个任务队列和N个可用的线程，由这些NIO线程负责消息的读取、解码、编码和发送。
- 一个NIO线程可以同时处理N条链路，但是一个链路只对应一个NIO线程，防止发生并发操作问题。

在绝大多数场景下，Reactor 多线程模型可以满足性能需求。但是，在个别特殊场景中，一个 NIO 线程负责监听和处理所有的客户端连接可能会存在性能问题。例如并发百万客户端连接，或者服务端需要对客户端握手进行安全认证，但是认证本身非常损耗性能。在这类场景下，单独一个 Acceptor 线程可能会存在性能不足的问题，为了解决性能问题，产生了第三种 Reactor 线程模型——主从 Reactor 多线程模型。

5.1.3 主从 Reactor 线程模型

主从 Reactor 线程模型的特点是：服务端用于接收客户端连接的不再是一个单独的 NIO 线程，而是一个独立的 NIO 线程池。Acceptor 接收到客户端 TCP

连接请求并处理完成后（可能包含接入认证等），将新创建的 SocketChannel 注册到 I/O 线程池（sub reactor 线程池）的某个 I/O 线程上，由它负责 SocketChannel 的读写和编解码工作。Acceptor 线程池仅仅用于客户端的登录、握手和安全认证，一旦链路建立成功，就将链路注册到后端 subReactor 线程池的 I/O 线程上，由 I/O 线程负责后续的 I/O 操作。

它的线程模型如图 5-3 所示。

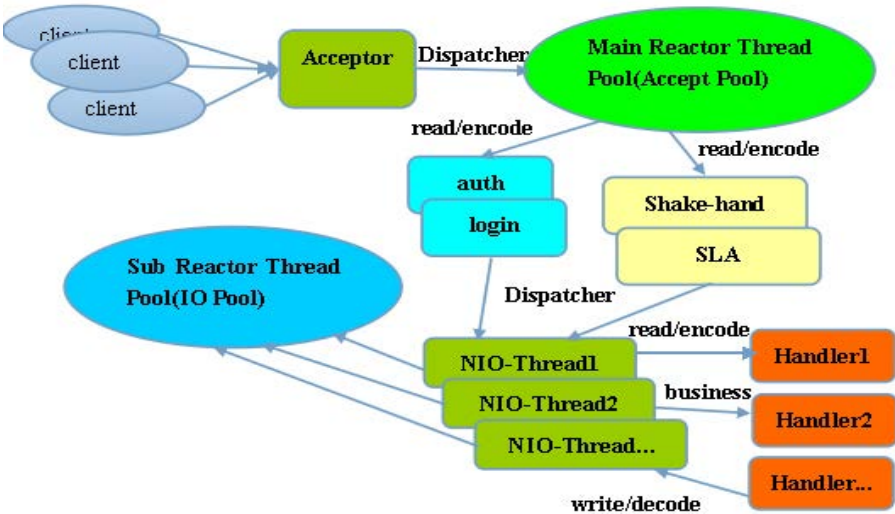


图 5-3 主从 Reactor 多线程模型

利用主从 NIO 线程模型，可以解决一个服务端监听线程无法有效处理所有客户端连接的性能不足问题。因此，在 Netty 的官方 demo 中，推荐使用该线程模型。

5.2 Netty 线程模型

Netty 的线程模型并不是一成不变的，它实际取决于用户的启动参数配置。通过设置不同的启动参数，Netty 可以同时支持 Reactor 单线程模型、多线程模型和主从 Reactor 多线程模型。

下面让我们通过一张原理图（图 5-4）来快速了解 Netty 的线程模型：

可以通过调整 Netty 服务端启动参数来设置它的线程模型。

服务端启动的时候，创建了两个 NioEventLoopGroup，它们实际是两个独立的 Reactor 线程池。一个用于接收客户端的 TCP 连接，另一个用于处理 I/O 相关

的读写操作，或者执行系统 Task、定时任务 Task 等。

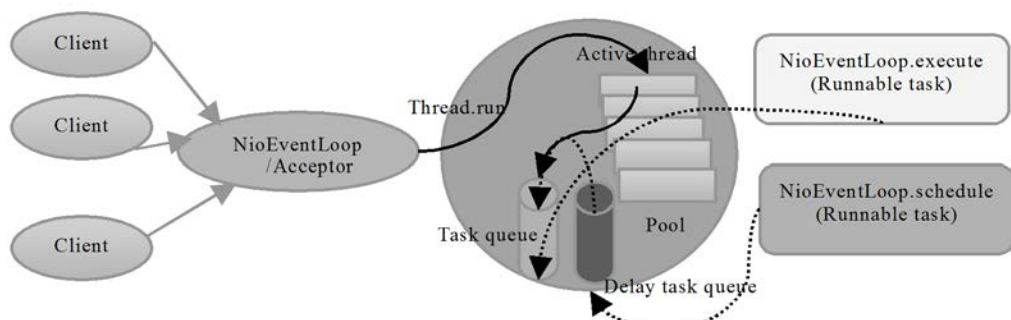


图 5-4 Netty 的线程模型

```
EventLoopGroup bossGroup = new NioEventLoopGroup(1);
EventLoopGroup workerGroup = new NioEventLoopGroup(2);
EventLoopGroup group = new NioEventLoopGroup();
try {
    ServerBootstrap b = new ServerBootstrap();
    b.group(bossGroup, workerGroup)
      .channel(NioServerSocketChannel.class)
```

Netty 用于接收客户端请求的线程池职责如下。

- (1) 接收客户端 TCP 连接，初始化 Channel 参数；
- (2) 将链路状态变更事件通知给 ChannelPipeline。

Netty 处理 I/O 操作的 Reactor 线程池职责如下。

- (1) 异步读取通信对端的数据报，发送读事件到 ChannelPipeline；
- (2) 异步发送消息到通信对端，调用 ChannelPipeline 的消息发送接口；
- (3) 执行系统调用 Task；
- (4) 执行定时任务 Task，例如链路空闲状态监测定时任务。

通过调整线程池的线程个数、是否共享线程池等方式，Netty 的 Reactor 线程模型可以在单线程、多线程和主从多线程间切换，这种灵活的配置方式可以最大程度地满足不同用户的个性化定制。

为了尽可能地提升性能，Netty 在很多地方进行了无锁化的设计，例如在 I/O 线程内部进行串行操作，避免多线程竞争导致的性能下降问题。表面上看，串

行化设计似乎 CPU 利用率不高，并发程度不够。但是，通过调整 NIO 线程池的线程参数，可以同时启动多个串行化的线程并行运行，这种局部无锁化的串行线程设计相比一个队列—多个工作线程的模型性能更优。

它的设计原理如图 5-5 所示：

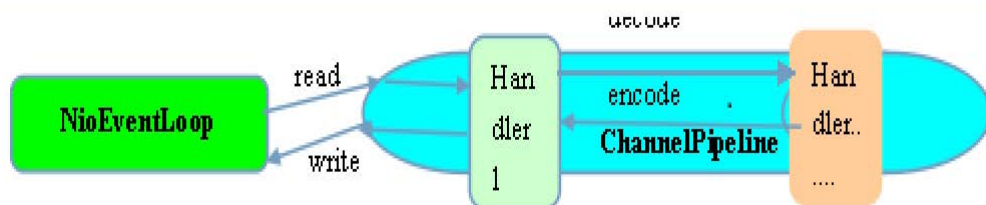


图 5-5 Netty 的设计原理

Netty 的 NioEventLoop 读取到消息之后，直接调用 ChannelPipeline 的 fireChannelRead (Object msg)。只要用户不主动切换线程，一直都是由 NioEventLoop 调用用户的 Handler，期间不进行线程切换。这种串行化处理方式避免了多线程操作导致的锁的竞争，从性能角度看是最优的。

5.3 最佳实践

5.3.1 时间可控的简单业务直接在 I/O 线程上处理

时间可控的简单业务直接在 I/O 线程上处理，如果业务非常简单，执行时间非常短，不需要与外部网元交互、访问数据库和磁盘，不需要等待其它资源，则建议直接在业务 ChannelHandler 中执行，不需要再启业务的线程或者线程池。避免线程上下文切换，也不存在线程并发问题。

5.3.2 复杂和时间不可控业务建议投递到后端业务线程池统一处理

复杂和时间不可控业务建议投递到后端业务线程池统一处理，对于此类业务，不建议直接在业务 ChannelHandler 中启动线程或者线程池处理，建议将不同的业务统一封装成 Task，统一投递到后端的业务线程池中进行处理。过多的业务 ChannelHandler 会带来开发效率和可维护性问题，不要把 Netty 当作业务容器，对于大多数复杂的业务产品，仍然需要集成或者开发自己的业务容器，做好和 Netty 的架构分层。

5.3.3 业务线程避免直接操作 ChannelHandler

业务线程避免直接操作 ChannelHandler，对于 ChannelHandler，IO 线程和业务线程都可能会操作，因为业务通常是多线程模型，这样就会存在多线程操作 ChannelHandler。为了尽量避免多线程并发问题，建议按照 Netty 自身的做法，通过将操作封装成独立的 Task 由 NioEventLoop 统一执行，而不是业务线程直接操作，相关代码如下所示：

```
if (ctx.executor().inEventLoop()) {  
    try {  
        doFlush(ctx);  
    } catch (Exception e) {  
        if (logger.isWarnEnabled()) {  
            logger.warn("Unexpected exception while sending chunks.", e);  
        }  
    }  
} else {  
    // let the transfer resume on the next event loop round  
    ctx.executor().execute(new Runnable() {  
  
        @Override  
        public void run() {  
            try {  
                doFlush(ctx);  
            } catch (Exception e) {  

```

如果你确认并发访问的数据或者并发操作是安全的，则无需多此一举，这个需要根据具体的业务场景进行判断，灵活处理。

5.3.4 线程数量计算

推荐的线程数量计算公式有以下两种。

- 公式一：线程数量=（线程总时间/瓶颈资源时间）× 瓶颈资源的线程并行数；
- 公式二：QPS=1000/线程总时间×线程数。

由于用户场景的不同，对于一些复杂的系统，实际上很难计算出最优线程配置，只能是根据测试数据和用户场景，结合公式给出一个相对合理的范围，然后对范围内的数据进行性能测试，选择相对最优值。

6 Netty 架构剖析

6.1 逻辑架构

Netty 采用了典型的三层网络架构进行设计和开发，逻辑架构如图 6-1 所示：

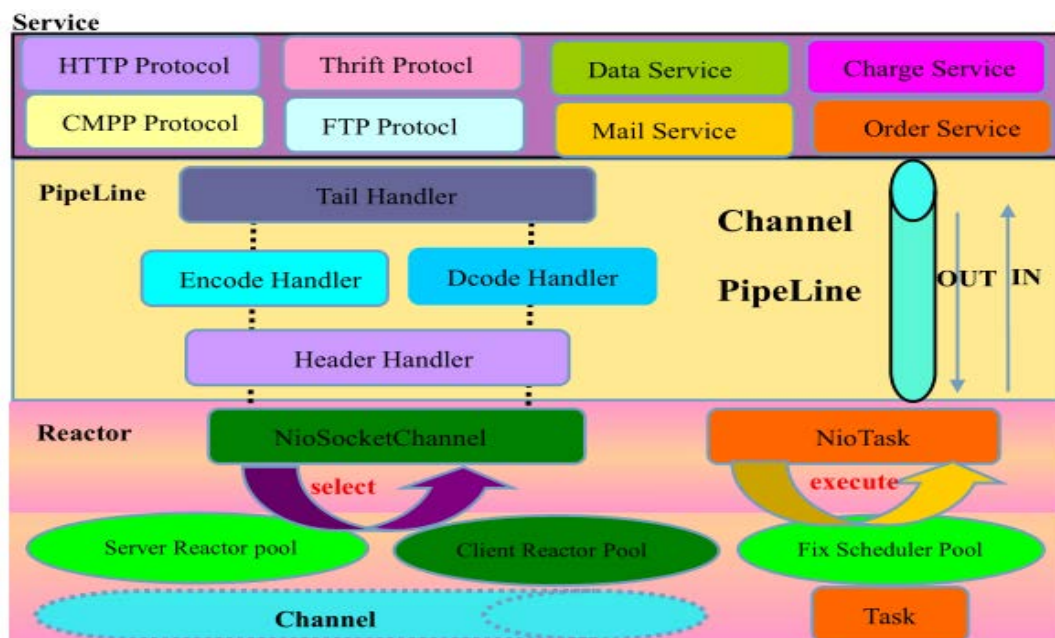


图 6-1 Netty 逻辑架构图

Reactor 通信调度层：它由一系列辅助类完成，包括 Reactor 线程 `NioEventLoop` 及其父类，`NioSocketChannel/ NioServerSocketChannel` 及其父类，`ByteBuffer` 以及由其衍生出来的各种 `Buffer`，`Unsafe` 以及其衍生出的各种内部类等。该层的主要职责就是监听网络的读写和连接操作，负责将网络层的数据读取到内存缓冲区中，然后触发各种网络事件，例如连接创建、连接激活、读事件、写事件等，将这些事件触发到 `PipeLine` 中，由 `PipeLine` 管理的职责链来进行后续的处理。

职责链 `ChannelPipeline`：它负责事件在职责链中的有序传播，同时负责动态地编排职责链。职责链可以选择监听和处理自己关心的事件，它可以拦截处理和向后 / 向前传播事件。不同应用的 `Handler` 节点的功能也不同，通常情况下，往往会开发编解码 `Handler` 用于消息的编解码，它可以将外部的协议消息转换成内部的 `POJO` 对象，这样上层业务则只需要关心处理业务逻辑即可，不需要感知底层的协议差异和线程模型差异，实现了架构层面的分层隔离。

业务逻辑编排层 (`Service ChannelHandler`)：业务逻辑编排层通常有两类：一类是纯粹的业务逻辑编排，还有一类是其他的应用层协议插件，用于特定协议相关的会话和链路管理。例如 `CMPP` 协议，用于管理和中国移动短信系统的对接。

架构的不同层面，需要关心和处理的对象都不同，通常情况下，对于业务开发者，只需要关心职责链的拦截和业务 `Handler` 的编排。因为应用层协议栈往往是开发一次，到处运行，所以实际上对于业务开发者来说，只需要关心服务层的业务逻辑开发即可。各种应用协议以插件的形式提供，只有协议开发人员需要关注协议插件，对于其他业务开发人员来说，只需关心业务逻辑定制。这种分层的架构设计理念实现了 `NIO` 框架各层之间的解耦，便于上层业务协议栈的开发和业务逻辑的定制。

正是由于 `Netty` 的分层架构设计非常合理，基于 `Netty` 的各种应用服务器和协议栈开发才能够如雨后春笋般得到快速发展。

6.2 关键架构质量属性

6.2.1 高性能

影响最终产品的性能因素非常多，其中软件因素如下：

- 架构不合理导致的性能问题；
- 编码实现不合理导致的性能问题，例如锁的不恰当使用导致性能瓶颈。

硬件因素如下：

- 服务器硬件配置太低导致的性能问题；
- 带宽、磁盘的IOPS等限制导致的I/O操作性能差；
- 测试环境被共用导致被测试的软件产品受到影响。

尽管影响产品性能的因素非常多，但是架构的性能模型合理与否对性能的影响非常大。如果一个产品的架构设计得不好，无论开发如何努力，都很难开发出一个高性能、高可用的软件产品。

“性能是设计出来的，而不是测试出来的”。下面我们看 Netty 的架构设计是如何实现高性能的。

(1) 采用异步非阻塞的 I/O 类库，基于 Reactor 模式实现，解决了传统同步阻塞 I/O 模式下一个服务端无法平滑地处理线性增长的客户端的问题。

(2) TCP 接收和发送缓冲区使用直接内存代替堆内存，避免了内存复制，提升了 I/O 读取和写入的性能。

(3) 支持通过内存池的方式循环利用 ByteBuf，避免了频繁创建和销毁 ByteBuf 带来的性能损耗。

(4) 可配置的 I/O 线程数、TCP 参数等，为不同的用户场景提供定制化的调优参数，满足不同的性能场景。

(5) 采用环形数组缓冲区实现无锁化并发编程，代替传统的线程安全容器或者锁。

(6) 合理地使用线程安全容器、原子类等，提升系统的并发处理能力。

(7) 关键资源的处理使用单线程串行化的方式，避免多线程并发访问带来的锁竞争和额外的 CPU 资源消耗问题。

(8) 通过引用计数器及时地申请释放不再被引用的对象，细粒度的内存管理降低了 GC 的频率，减少了频繁 GC 带来的时延增大和 CPU 损耗。

无论是 Netty 的官方性能测试数据，还是携带业务实际场景的性能测试，Netty 在各个 NIO 框架中综合性能是最高的。

6.2.2 可靠性

作为一个高性能的异步通信框架，架构的可靠性是大家选择的一个重要依据。下面我们探讨 Netty 架构的可靠性设计。

1. 链路有效性检测

由于长连接不需要每次发送消息都创建链路，也不需要消息交互完成时关闭链路，因此相对于短连接性能更高。对于长连接，一旦链路建立成功便一直维系双方之间的链路，直到系统退出。

为了保证长连接的链路有效性，往往需要通过心跳机制周期性地链路检测。使用周期性心跳的原因是：在系统空闲时，例如凌晨，往往没有业务消息。如果此时链路被防火墙 Hang 住，或者遭遇网络闪断、网络单通等，通信双方无法识别出这类链路异常。等到第二天业务高峰期到来时，瞬间的海量业务冲击会导致消息积压无法发送给对方，由于链路的重建需要时间，这期间业务会大量失败（集群或者分布式组网情况会好一些）。为了解决这个问题，需要周期性的心跳对链路进行有效性检测，一旦发生问题，可以及时关闭链路，重建 TCP 连接。

当有业务消息时，无须心跳检测，可以由业务消息进行链路可用性检测。所以心跳消息往往是在链路空闲时发送的。

为了支持心跳，Netty 提供了如下两种链路空闲检测机制：

- 读空闲超时机制：当连续周期T没有消息可读时，触发超时Handler，用户可以基于读空闲超时发送心跳消息，进行链路检测；如果连续N个周期仍然没有读取到心跳消息，可以主动关闭链路。

- 写空闲超时机制：当连续周期T没有消息要发送时，触发超时Handler，用户可以基于写空闲超时发送心跳消息，进行链路检测；如果连续N个周期仍然没有接收到对方的心跳消息，可以主动关闭链路。

为了满足不同用户场景的心跳定制，Netty 提供了空闲状态检测事件通知机制，用户可以订阅空闲超时事件、写空闲超时事件、读或者写超时事件，在接收到对应的空闲事件之后，灵活地进行定制。

2. 内存保护机制

Netty 提供多种机制对内存进行保护，包括以下几个方面：

- 通过对象引用计数器对Netty的ByteBuf等内置对象进行细粒度的内存申请和释放，对非法的对象引用进行检测和保护。
- 通过内存池来重用ByteBuf，节省内存。
- 可设置的内存容量上限，包括ByteBuf、线程池线程数等。

如果长度解码器没有单个消息最大报文长度限制，当解码错位或者读取到畸形码流时，长度值可能是个超大整数值，例如 4294967296，这很容易导致内存溢出。如果有上限保护，例如单条消息最大不允许超过 10MB，当读取到非法消息长度 4294967296 后，直接抛出解码异常，这样就避免了大内存的分配。

3. 优雅停机

相比于 Netty 的早期版本，Netty 5.0 版本的优雅退出功能做得更加完善。优雅停机功能指的是当系统退出时，JVM 通过注册的 Shutdown Hook 拦截到退出信号量，然后执行退出操作，释放相关模块的资源占用，将缓冲区的消息处理完成或者清空，将待刷新的数据持久化到磁盘或者数据库中，等到资源回收和缓冲区消息处理完成之后，再退出。

优雅停机往往需要设置个最大超时时间 T，如果达到 T 后系统仍然没有退出，则通过 Kill - 9 pid 强杀当前的进程。Netty 所有涉及到资源回收和释放的地方都增加了优雅退出的方法。它们的相关接口如表 6-1 所示。

6.2.3 可定制性

Netty 的可定制性主要体现在以下几点：

- 责任链模式：ChannelPipeline基于责任链模式开发，便于业务逻辑的拦截、定制和扩展。
- 基于接口的开发：关键的类库都提供了接口或者抽象类，如果Netty自身的实现无法满足用户的需求，可以由用户自定义实现相关接口。
- 提供了大量工厂类，通过重载这些工厂类可以按需创建出用户实现的对象。
- 提供了大量的系统参数供用户按需设置，增强系统的场景定制性。

EventExecutorGroup.shutdownGracefully()	NIO线程优雅退出
EventExecutorGroup.shutdownGracefully(long quietPeriod, long timeout, TimeUnit unit)	NIO线程优雅退出，支持设置超时时间
Channel.close()	Channel的关闭
Unsafe.close(ChannelPromise promise)	Unsafe的关闭操作，可以设置可写的Future
Unsafe.closeForcibly()	Unsafe的强制关闭操作
ChannelPipeline.close()	ChannelPipeline的关闭
ChannelPipeline.close(ChannelPromise promise)	ChannelPipeline的关闭，可以设置可写的Future
ChannelHandler.close(ChannelHandlerContext ctx, ChannelPromise promise)	ChannelHandler的关闭

表 6-1 Netty 重要资源的优雅退出方法

6.2.4 可扩展性

基于 Netty 的基础 NIO 框架，可以方便地进行应用层协议定制，例如 HTTP 协议栈、Thrift 协议栈、FTP 协议栈等。这些扩展不需要修改 Netty 的源码，直接基于 Netty 的二进制类库即可实现协议的扩展和定制。

目前，业界存在大量的基于 Netty 框架开发的协议，例如基于 Netty 的 HTTP 协议、Dubbo 协议、RocketMQ 内部私有协议等。

7 Netty 案例集锦

7.1 内存泄漏类

7.1.1 问题描述

业务代码升级 Netty 3 到 Netty4 之后，运行一段时间，Java 进程就会宕机，查看系统运行日志发现系统发生了内存泄露（示例堆栈）：

```
java.lang.OutOfMemoryError: Direct buffer memory
    at java.nio.Bits.reserveMemory(Bits.java:658)
    at java.nio.DirectByteBuffer.<init>(DirectByteBuffer.java:123)
    at java.nio.ByteBuffer.allocateDirect(ByteBuffer.java:306)
    at io.netty.buffer.PoolArena$DirectArena.newChunk(PoolArena.java:383)
    at io.netty.buffer.PoolArena.allocateNormal(PoolArena.java:143)
    at io.netty.buffer.PoolArena.allocate(PoolArena.java:132)
    at io.netty.buffer.PoolArena.allocate(PoolArena.java:94)
    at io.netty.buffer.PooledByteBufAllocator.newDirectBuffer(PooledByteBufAllocator.java:238)
    at io.netty.buffer.AbstractByteBufAllocator.directBuffer(AbstractByteBufAllocator.java:155)
    at io.netty.buffer.AbstractByteBufAllocator.directBuffer(AbstractByteBufAllocator.java:146)
    at io.netty.buffer.AbstractByteBufAllocator.ioBuffer(AbstractByteBufAllocator.java:107)
    at io.netty.example.echo.EchoClientHandler.channelRead(EchoClientHandler.java:77)
    at io.netty.channel.ChannelHandlerInvokerUtil.invokeChannelReadNow(ChannelHandlerInvokerUtil.java:74)
    at io.netty.channel.DefaultChannelHandlerInvoker.invokeChannelRead(DefaultChannelHandlerInvoker.java:138)
    at io.netty.channel.DefaultChannelHandlerContext.fireChannelRead(DefaultChannelHandlerContext.java:320)
    at io.netty.channel.DefaultChannelPipeline.fireChannelRead(DefaultChannelPipeline.java:846)
    at io.netty.channel.nio.AbstractNioByteChannel$NioByteUnsafe.read(AbstractNioByteChannel.java:127)
    at io.netty.channel.nio.NioEventLoop.processSelectedKey(NioEventLoop.java:485)
    at io.netty.channel.nio.NioEventLoop.processSelectedKeysOptimized(NioEventLoop.java:452)
    at io.netty.channel.nio.NioEventLoop.run(NioEventLoop.java:346)
    at io.netty.util.concurrent.SingleThreadEventExecutor$5.run(SingleThreadEventExecutor.java:794)
    at java.lang.Thread.run(Thread.java:745)
```

图 7-1 内存泄漏堆栈

对内存进行监控（切换使用堆内存池，方便对内存进行监控），发现堆内存一直飙升，如下所示（示例堆内存监控）：

这些是当前可用内存池

筛选	池名称					
池名称	类型	已用	最大值	使用量	已用峰值	最大值峰值
PS Eden Space	HEAP	4.53 MB	316.00 MB	1.43%	15.50 MB	322.00 MB
PS Survivor Space	HEAP	5.03 MB	5.50 MB	91.48%	5.03 MB	5.50 MB
Code Cache	NON_HEAP	1.45 MB	48.00 MB	3.02%	1.46 MB	48.00 MB
PS Perm Gen	NON_HEAP	16.63 MB	82.00 MB	20.29%	16.63 MB	82.00 MB
PS Old Gen	HEAP	586.28 MB	653.00 MB	89.78%	586.28 MB	653.00 MB

图 7-2 堆内存监控示例

7.1.2 问题定位

使用 `jmap -dump:format=b,file=netty.bin PID` 将堆内存 dump 出来，通过 IBM 的 HeapAnalyzer 工具进行分析，发现 ByteBuf 发生了泄露。

因为使用了 Netty 4 的内存池，所以首先怀疑是不是申请的 ByteBuf 没有被释放导致？查看代码，发现消息发送完成之后，Netty 底层已经调用 `ReferenceCountUtil.release(message)` 对内存进行了释放。这是怎么回事呢？难道 Netty 4.X 的内存池有 Bug，调用 `release` 操作释放内存失败？

考虑到 Netty 内存池自身 Bug 的可能性不大，首先从业务的使用方式入手分析。

内存的分配是在业务代码中进行，由于使用到了业务线程池做 I/O 操作和业务操作的隔离，实际上内存是在业务线程中分配的。

内存的释放操作是在 outbound 中进行，按照 Netty 3 的线程模型，downstream（对应 Netty 4 的 outbound，Netty 4 取消了 upstream 和 downstream）的 handler 也是由业务调用者线程执行的，也就是说申请和释放在同一个业务线程中进行。初次排查并没有发现导致内存泄露的根因，继续分析 Netty 内存池的实现原理。

Netty 内存池实现原理分析：查看 Netty 的内存池分配器 `PooledByteBu-`

fAllocator 的源码实现，发现内存池实际是基于线程上下文实现的，相关代码如下：

```
final ThreadLocal<PoolThreadCache> threadCache = new  
ThreadLocal<PoolThreadCache>() {  
    private final AtomicInteger index = new AtomicInteger();  
    @Override  
    protected PoolThreadCache initialValue() {  
        final int idx = index.getAndIncrement();  
        final PoolArena<byte[]> heapArena;  
        final PoolArena<ByteBuffer> directArena;  
        //.....此处代码省略  
        return new PoolThreadCache(heapArena, directArena);  
    }  
}
```

也就是说内存的申请和释放必须在同一线程上下文中，不能跨线程。跨线程之后实际操作的就不是同一块儿内存区域，这会导致很多严重的问题，内存泄露便是其中之一。内存在 A 线程申请，切换到 B 线程释放，实际是无法正确回收的。

7.1.3 问题根因

Netty 4 修改了 Netty 3 的线程模型：在 Netty 3 的时候，upstream 是在 I/O 线程里执行的，而 downstream 是在业务线程里执行。当 Netty 从网络读取一个数据报投递给业务 handler 的时候，handler 是在 I/O 线程里执行；而我们在业务线程中调用 write 和 writeAndFlush 向网络发送消息的时候，handler 是在业务线程里执行，直到最后一个 Header handler 将消息写入到发送队列中，业务线程才返回。

Netty4 修改了这一模型，在 Netty 4 里 inbound(对应 Netty 3 的 upstream) 和 outbound(对应 Netty 3 的 downstream) 都是在 NioEventLoop(I/O 线程) 中执行。当我们在业务线程里通过 ChannelHandlerContext.write 发送消息的时候，Netty 4 在将消息发送事件调度到 ChannelPipeline 的时候，首先将

待发送的消息封装成一个 Task，然后放到 NioEventLoop 的任务队列中，由 NioEventLoop 线程异步执行。后续所有 handler 的调度和执行，包括消息的发送、I/O 事件的通知，都由 NioEventLoop 线程负责处理。

在本案例中，ByteBuf 在业务线程中申请，在后续的 ChannelHandler 中释放，ChannelHandler 是由 Netty 的 I/O 线程 (EventLoop) 执行的，因此内存的申请和释放不在同一个线程中，导致内存泄漏。

Netty 3 的 I/O 事件处理流程：

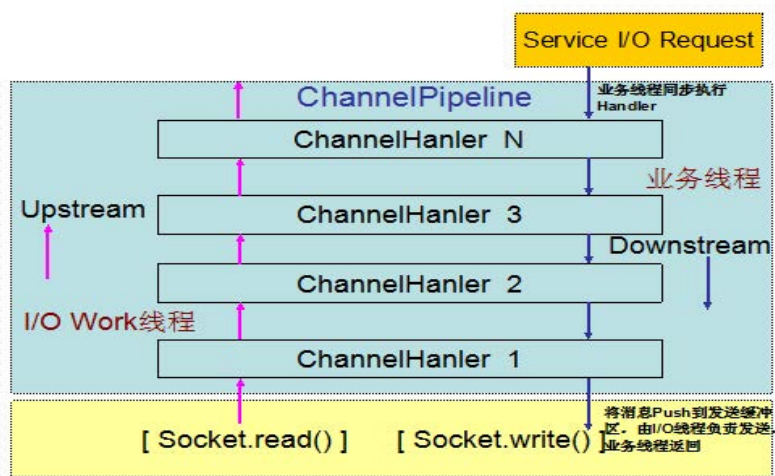


图 7-3 Netty 3 的 I/O 线程模型

Netty 4 的 I/O 消息处理流程：

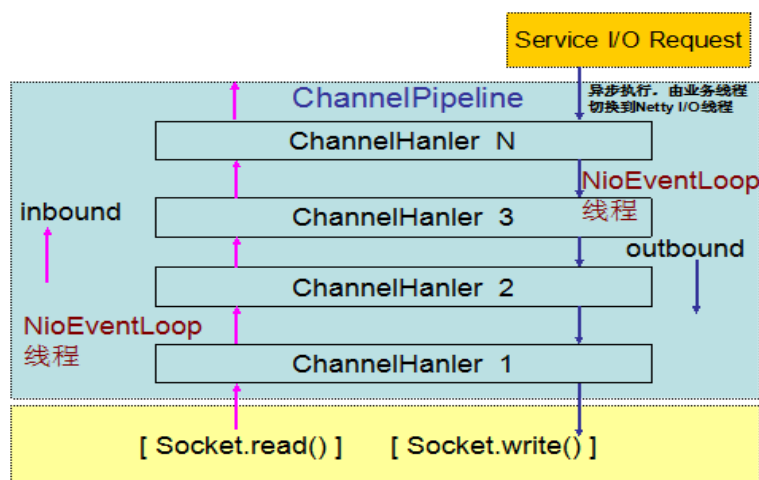


图 7-4 Netty 4 I/O 线程模型

7.1.4 案例总结

Netty 4.X 版本新增的内存池确实非常高效，但是如果使用不当则会导致各种严重的问题。诸如内存泄露这类问题，功能测试并没有异常，如果相关接口没有进行压测或者稳定性测试而直接上线，则会导致严重的线上问题。

内存池 PooledByteBuf 的使用建议。

- 申请之后一定要记得释放，Netty 自身 Socket 读取和发送的 ByteBuf 系统会自动释放，用户不需要做二次释放；如果用户使用 Netty 的内存池在应用中做 ByteBuf 的对象池使用，则需要自己主动释放。
- 避免错误的释放：跨线程释放、重复释放等都是非法操作，要避免。特别是跨线程申请和释放，往往具有隐蔽性，问题定位难度较大。
- 防止隐式的申请和分配：之前曾经发生过一个案例，为了解决内存池跨线程申请和释放问题，有用户对内存池做了二次包装，以实现多线程操作时，内存始终由包装的管理线程申请和释放，这样可以屏蔽用户业务线程模型和访问方式的差异。谁知运行一段时间之后再次发生了内存泄露，最后发现原来调用 ByteBuf 的 write 操作时，如果内存容量不足，会自动进行容量扩展。扩展操作由业务线程执行，这就绕过了内存池管理线程，发生了“引用逃逸”。
- 避免跨线程申请和使用内存池，由于存在“引用逃逸”等隐式的内存创建，实际上跨线程申请和使用内存池是非常危险的行为。尽管从技术角度看可以实现一个跨线程协调的内存池机制，甚至重写 PooledByteBufAllocator，但是这无疑会增加很多复杂性，通常也使用不到。如果确实存在跨线程的 ByteBuf 传递，而且无法保证 ByteBuf 在另一个线程中会重新分配大小等操作，最简单保险的方式就是在线程切换点做一次 ByteBuf 的拷贝，但这会造成性能下降。

比较好的一种方案就是如果存在跨线程的 ByteBuf 传递，对 ByteBuf 的写操作要在分配线程完成，另一个线程只能做读操作。操作完成之后发送一个事件通

知分配线程，由分配线程执行内存释放操作。

7.2 性能瓶颈类

7.2.1 问题描述

业务代码升级 Netty 3 到 Netty4 之后，并没有给产品带来预期的性能提升，有些甚至还发生了非常严重的性能下降，这与 Netty 官方给出的数据并不一致。

Netty 官方性能测试对比数据：我们比较了两个分别建立在 Netty 3 和 4 基础上 echo 协议服务器。（Echo 非常简单，这样，任何垃圾的产生都是 Netty 的原因，而不是协议的原因）。我使它们服务于相同的分布式 echo 协议客户端，来自这些客户端的 16384 个并发连接重复发送 256 字节的随机负载，几乎使千兆以太网饱和。

根据测试结果，Netty 4:

- GC中断频率是原来的1/5： 45.5 vs. 9.2次/分钟
- 垃圾生成速度是原来的1/5： 207.11 vs 41.81 MiB/秒

7.2.2 问题定位

首先通过 JMC 等性能分析工具对性能热点进行分析，示例如下（信息安全等原因，只给出分析过程示例截图）：

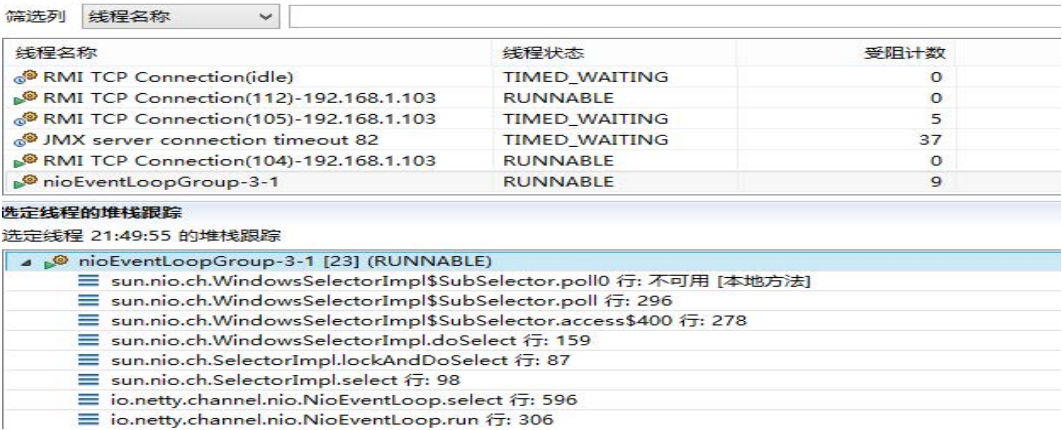


图 7-5 性能热点线程堆栈

通过对热点方法的分析，发现在消息发送过程中，有两处热点：

- 消息发送性能统计相关Handler；
- 编码Handler。

对使用 Netty 3 版本的业务产品进行性能对比测试，发现上述两个 Handler 也是热点方法。既然都是热点，为啥切换到 Netty4 之后性能下降这么厉害呢？

通过方法的调用树分析发现了两个版本的差异：在 Netty 3 中，上述两个热点方法都是由业务线程负责执行；而在 Netty 4 中，则是由 NioEventLoop(I/O) 线程执行。对于某个链路，业务是拥有多个线程的线程池，而 NioEventLoop 只有一个，所以执行效率更低，返回给客户端的应答时延就大。时延增大之后，自然导致系统并发量降低，性能下降。

找出问题根因之后，针对 Netty 4 的线程模型对业务进行专项优化，将耗时的编码等操作迁移到业务线程中执行，为 I/O 线程减负，性能达到预期，远超过了 Netty 3 老版本的性能。

Netty 3 的业务线程调度模型图如下所示：充分利用了业务多线程并行编码和 Handler 处理的优势，周期 T 内可以处理 N 条业务消息：

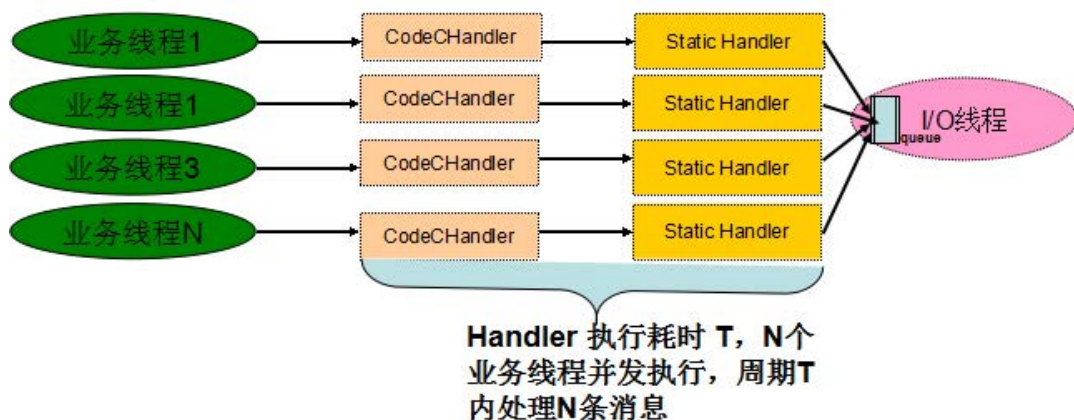


图 7-6 Netty 3 Handler 执行线程模型

切换到 Netty 4 之后，业务耗时 Handler 被 I/O 线程串行执行，因此性能发生比较大的下降：

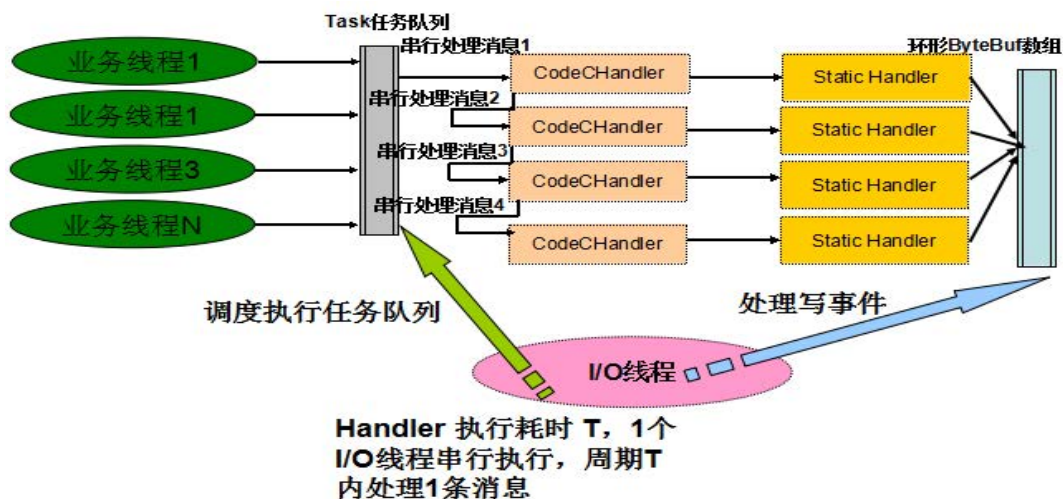


图 7-7 Netty 4 Handler 执行线程模型

7.2.3 问题总结

该问题的根因还是由于 Netty 4 的线程模型变更引起，线程模型变更之后，不仅影响业务的功能，甚至对性能也会造成很大的影响。

对 Netty 的升级需要从功能、兼容性和性能等多个角度进行综合考虑，切不可只盯着 API 变更这个芝麻，而丢掉了性能这个西瓜。API 的变更会导致编译错误，但是性能下降却隐藏于无形之中，稍不注意就会中招。

对于讲究快速交付、敏捷开发和灰度发布的互联网应用，升级的时候更应该要当心。

7.3 线程膨胀类

7.3.1 问题描述

分布式服务框架在进行现网问题定位时，Dump 线程堆栈之后发现 Netty 的 NIO 线程竟然有 3000 多个，大量的 NIO 线程占用了系统的句柄资源、内存资源、CPU 资源等，引发了一些其它问题，需要尽快查明原因并解决线程过多问题。

7.3.2 问题分析

在研发环境中模拟现网组网和业务场景，使用 jmc 工具进行问题定位，使用飞行记录器对系统运行状况做快照，模拟示例图如下所示：

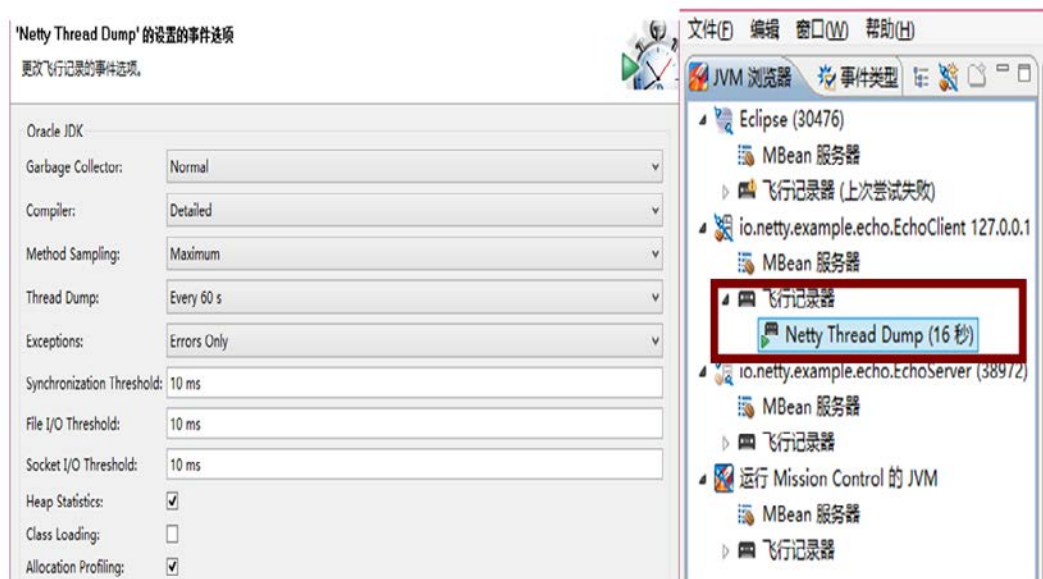


图 7-8 使用 jmc 工具进行问题定位

获取到黑匣子数据之后，可以对系统的各种重要指标做分析，包括系统数据、内存、GC 数据、线程运行状态和数据等。通过对线程堆栈分析，我们发现 Netty 的 NioEventLoop 线程超过了 3000 个！

活动线程

活动线程 22:17:37

筛选

线程名称

nioEventLoopGroup

☐ CPU 概要分析

☒ 死锁检测

☒ 分析

线程名称	线程状态	受阻计数	CPU 总体占用率	死锁	已分配的字节
 nioEventLoopGroup-3001-1	RUNNABLE	0	未启用	否	3.95 KB
 nioEventLoopGroup-3000-1	RUNNABLE	0	未启用	否	12.20 KB
 nioEventLoopGroup-2999-1	RUNNABLE	0	未启用	否	9.61 KB
 nioEventLoopGroup-2998-1	RUNNABLE	0	未启用	否	7.02 KB
 nioEventLoopGroup-2997-1	RUNNABLE	0	未启用	否	7.30 KB
 nioEventLoopGroup-2996-1	RUNNABLE	0	未启用	否	4.80 KB
 nioEventLoopGroup-2995-1	RUNNABLE	0	未启用	否	6.64 KB
 nioEventLoopGroup-2994-1	RUNNABLE	0	未启用	否	4.02 KB
 nioEventLoopGroup-2993-1	RUNNABLE	0	未启用	否	9.45 KB
 nioEventLoopGroup-2992-1	RUNNABLE	0	未启用	否	5.58 KB
 nioEventLoopGroup-2991-1	RUNNABLE	0	未启用	否	6.02 KB

选定线程的堆栈跟踪

图 7-9 Netty 线程占用超过 3000 个

对服务框架协议栈的 Netty 客户端和服务端源码进行 CodeReview，发现了

问题所在：

客户端每连接 1 个服务端，就会创建 1 个新的 `NioEventLoopGroup`，并设置它的线程数为 1。

现网有 300 个 + 节点，节点之间采用多链路（10 个链路），由于业务采用了随机路由，最终每个消费者需要跟其它 200 多个节点建立长连接，加上自己服务端也需要占用一些 `NioEventLoop` 线程，最终客户端单进程线程数膨胀到了 3000 多个。

业务的伪代码如下：

```
for(Link linkE : links)
{
    EventLoopGroup group = new NioEventLoopGroup(1);
    Bootstrap b = new Bootstrap();
    b.group(group)
      .channel(NioSocketChannel.class)
      .option(ChannelOption.TCP_NODELAY, true)

    // 此处省略 .....

    b.connect(linkE.localAddress, linkE.remoteAddress);
}
```

如果客户端对每个链路连接都创建一个新的 `NioEventLoopGroup`，则每个链路就会占用 1 个独立的 NIO 线程，最终沦为 1 连接：1 线程 这种同步阻塞模式线程模型。随着集群组网规模的不断扩大，这会带来严重的线程膨胀问题，最终会发生句柄耗尽无法创建新的线程，或者栈内存溢出。

从另一个角度看，1 个 NIO 线程只处理一条链路也体现不出非阻塞 I/O 的优势。案例中的错误线程模型如下所示。

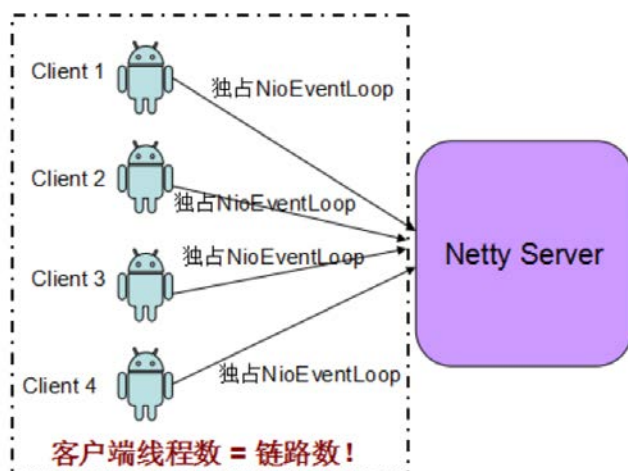


图 7-9 错误的客户端连接线程使用方式

7.3.3 案例总结

无论是服务端监听多个端口，还是客户端连接多个服务端，都需要注意必须要重用 NIO 线程，否则就会导致线程资源浪费，在大规模组网时还会存在句柄耗尽或者栈溢出等问题。

Netty 官方 Demo 仅仅是个 Sample，对用户而言，必须理解 Netty 的线程模型，否则很容易按照官方 Demo 的做法，在外层套个 For 循环连接多个服务端，然后，悲剧就这样发生了。

修正案例中的问题非常简单，原理如下：

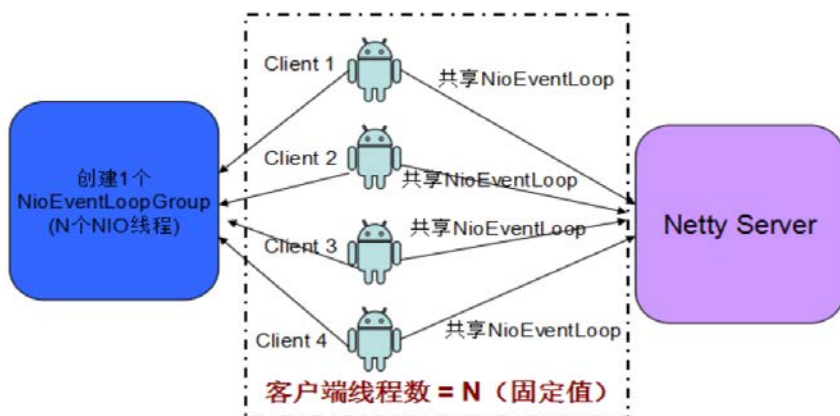


图 7-10 正确的客户端连接线程模型

7.4 客户端连接类

7.4.1 问题描述

Netty 客户端想同时连接多个服务端，使用如下方式，是否可行，我简单测试了下，暂时没有发现问题。代码如下：

```
EventLoopGroup group = new NioEventLoopGroup();

try {

    Bootstrap b = new Bootstrap();

    b.group(group)

    ..... 代码省略

    // Start the client.

    ChannelFuture f1 = b.connect(HOST, PORT);

    ChannelFuture f2 = b.connect(HOST2, PORT2);

    // Wait until the connection is closed.

    f1.channel().closeFuture().sync();

    f2.channel().closeFuture().sync();

    ..... 代码省略

}
```

7.4.2 答疑解惑

上述代码没有问题，原因是尽管 Bootstrap 自身不是线程安全的，但是执行 Bootstrap 的连接操作是串行执行的，而且 connect(String inetHost, int inetPort) 方法本身是线程安全的，它会创建一个新的 NioSocketChannel，并从初始构造的 EventLoopGroup 中选择一个 NioEventLoop 线程执行真正的 Channel 连接操作，与执行 Bootstrap 的线程无关，所以通过一个 Bootstrap 连续发起多个连接操作是安全的，它的原理如下：

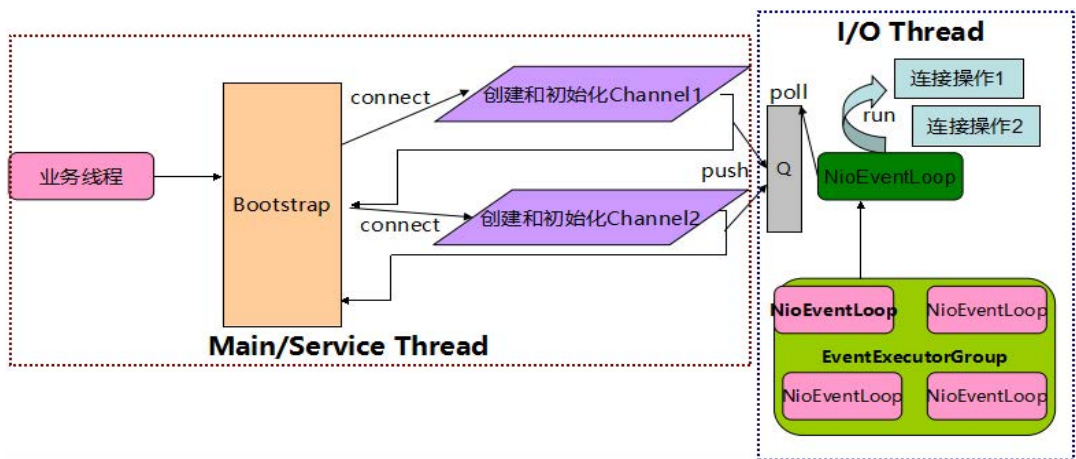


图 7-11 Netty BootStrap 工作原理

7.4.3 问题总结

注意事项 - 资源释放问题：在同一个 Bootstrap 中连续创建多个客户端连接，需要注意的是 EventLoopGroup 是共享的，也就是说这些连接共用一个 NIO 线程组 EventLoopGroup，当某个链路发生异常或者关闭时，只需要关闭并释放 Channel 本身即可，不能同时销毁 Channel 所使用的 NioEventLoop 和所在的线程组 EventLoopGroup，例如下面的代码片段就是错误的：

```
ChannelFuture f1 = b.connect(HOST, PORT);

ChannelFuture f2 = b.connect(HOST2, PORT2);

f1.channel().closeFuture().sync();

} finally {

    group.shutdownGracefully();

}
```

线程安全问题：需要指出的是 Bootstrap 不是线程安全的，因此在多个线程中并发操作 Bootstrap 是一件非常危险的事情，Bootstrap 是 I/O 操作工具类，它自身的逻辑处理非常简单，真正的 I/O 操作都是由 EventLoop 线程负责的，所以通常多线程操作同一个 Bootstrap 实例也是没有意义的，而且容易出错，错误代码如下：

```
Bootstrap b = new Bootstrap();  
  
{  
    // 多线程执行初始化、连接等操作  
}
```

7.5 线程安全类

7.5.1 问题描述

我有一个非线程安全的类 `ThreadUnsafeClass`，这个类会在 `channelRead` 方法中被调用。我下面这样的调用方法在多线程环境下安全吗？

代码示例如下：

```
public class MyHandler extends ChannelInboundHandlerAdapter {  
    private ThreadUnsafeClass unsafe = new ThreadUnsafeClass();  
    public void channelRead(ChannelHandlerContext ctx, Object  
msg) {  
        // 下面的代码是否 ok ?  
        unsafe.doSomething(ctx, msg);  
    }  
    .....  
}
```

7.5.2 答疑解惑

Netty 4 优化了 Netty 3 的线程模型，其中一个非常大的优化就是用户不需要再担心 `ChannelHandler` 会被并发调用，总结如下：

- `ChannelHandler`'s 的方法不会被 Netty 并发调用；
- 用户不再需要对 `ChannelHandler` 的各个方法做同步保护；
- `ChannelHandler` 实例不允许被多次添加到 `ChannelPipeline` 中，否则线程安全将得不到保证。

根据上述分析，`MyHandler` 的 `channelRead` 方法不会被并发调用，因此不存在线程安全问题。

7.5.3 问题总结

ChannelHandler 的线程安全存在几个特例，总结如下：

- 如果ChannelHandler被注解为 @Sharable，全局只有一个handler实例，它会被多个Channel的Pipeline共享，会被多线程并发调用，因此它不是线程安全的；
- 如果存在跨ChannelHandler的实例级变量共享，需要特别注意，它可能不是线程安全的。

非线程安全的跨 ChannelHandler 变量原理如下：

场景 1：串行调用，线程安全：

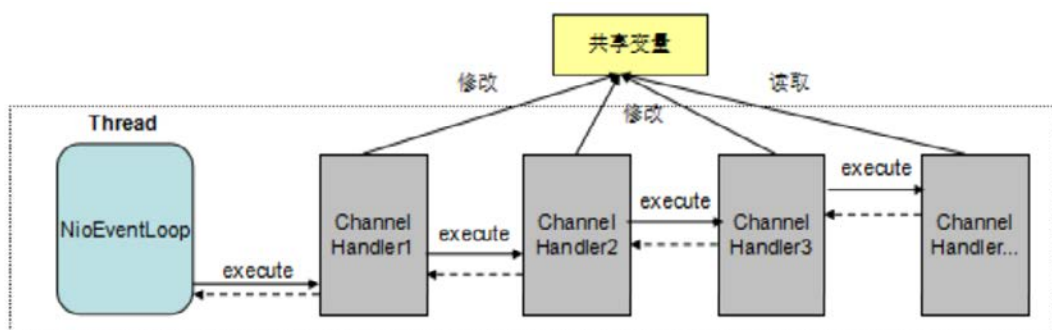


图 7-12 串行调用，线程安全

场景 2：并行调用，跨 Handler 共享变量，非线程安全：

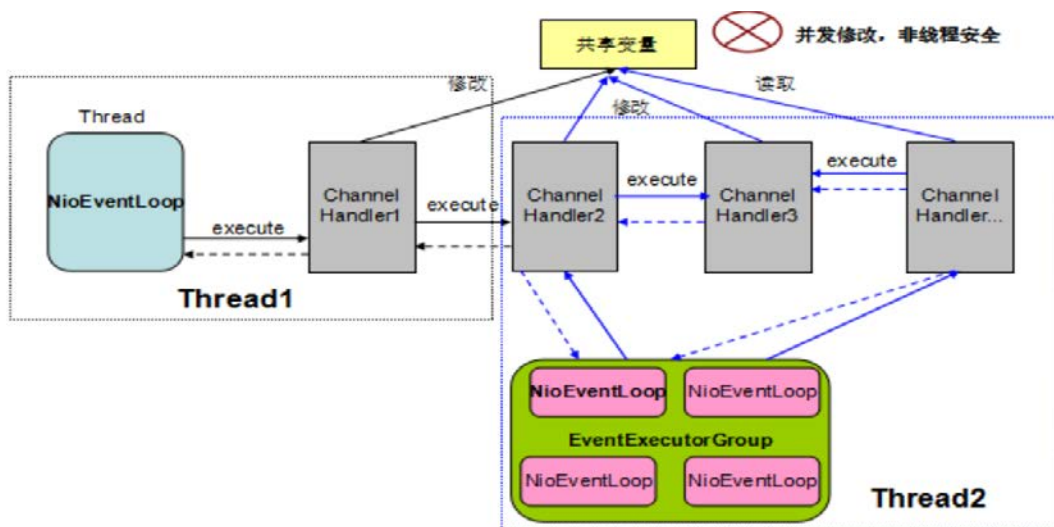


图 7-13 并行调用，非线程安全

7.6 消息接收类

7.6.1 问题描述

车联网服务端使用 Netty 构建，接收车载终端的请求消息，然后下发给后端其它系统，最后返回应答给车载终端。系统运行一段时间后发现服务端接收不到车载终端消息，需要尽快定位出问题原因。

7.6.2 问题定位

首先查看服务端进程运行状态、JVM 堆内存占用、CPU 使用率、网络 I/O 等，各种资源占用率都不高，排除了系统压力过大导致请求消息无法及时处理的因素。系统资源占用示例图：

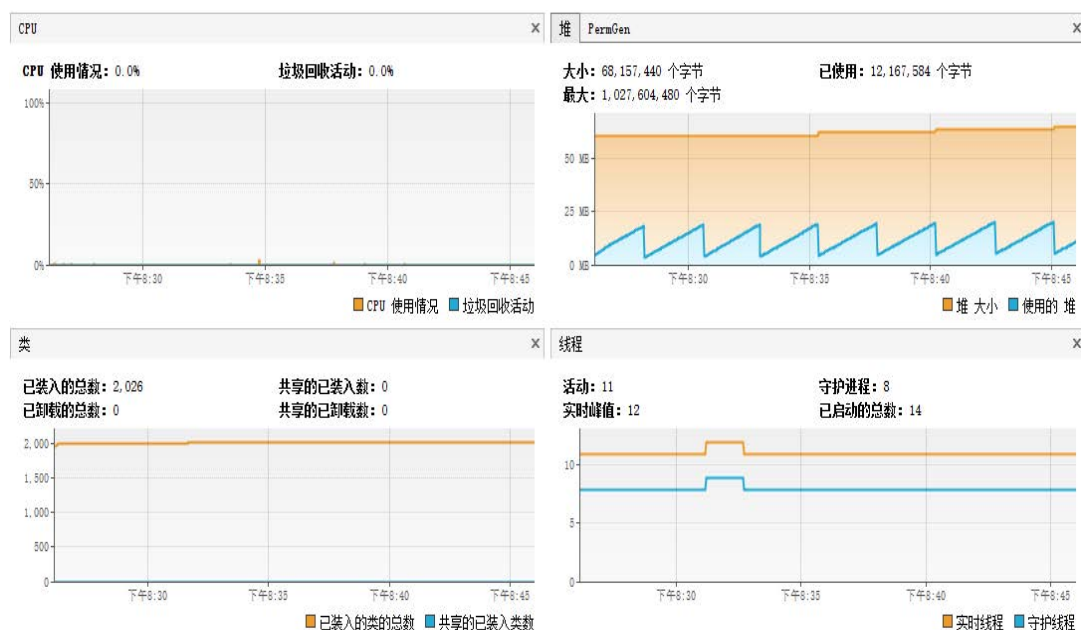


图 7-14 进程运行状态

排除资源耗尽等原因之外，可能导致无法接收请求消息的原因如下：

1. JVM 发生了长时间Full GC，导致进程暂停；
2. Netty的NIO线程跑飞或者被意外阻塞，导致无法接收请求消息；
3. Netty 解码发生了异常，导致请求ByteBuffer被缓存或者丢弃，无法解码到正常的业务消息；
4. 其它异常。

按照顺序我们进行问题排查，首先查看 Full GC 对消息接收的影响，通过对 GC 的监控统计，发上故障期间，系统并没有发生 Full GC，如下图所示。

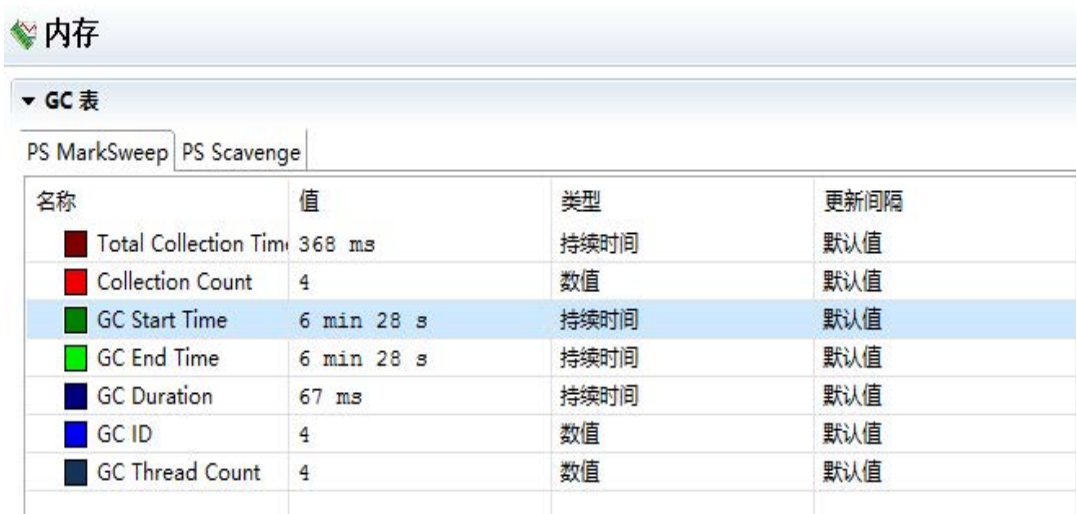


图 7-15 GC 状态

排除 Full GC 故障之后，继续排查 Netty 的 NIO 线程，采集故障发生时线程的工作状态，如下图所示：NioEventLoop 3 号线程组的 1 号线程被阻塞：

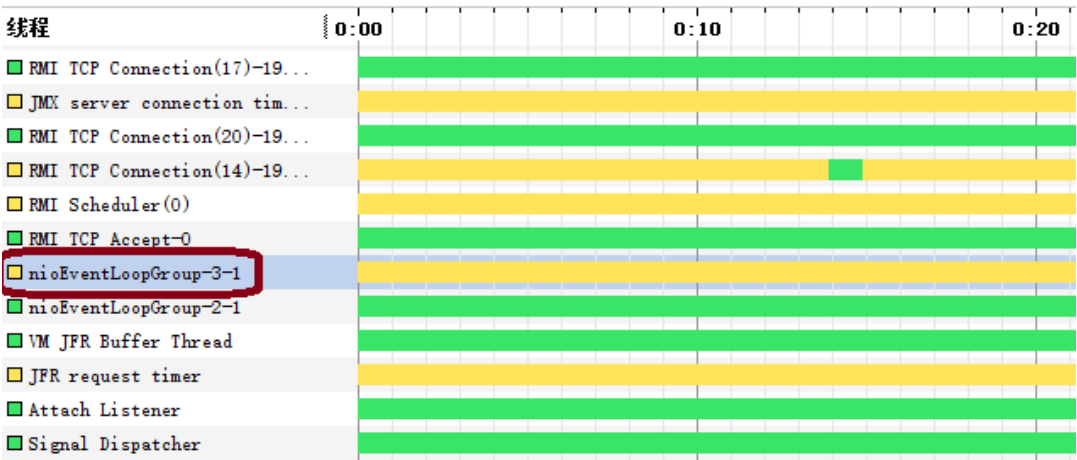


图 7-16 线程运行概括

原来是当转发给下游系统发生某些故障时，会导致业务定义的阻塞队列无法弹出消息进行处理，当队列积压满之后，就会阻塞 Netty 的 NIO 线程，而且无法自动恢复。

定位出问题原因之后，对 BUG 进行修复，故障排除。

线程	运行 ▼	休眠	等待	监视
RMI TCP Accept-0	3:27.259 (100.0%)	0.0 (0.0%)	0.0 (0.0%)	
nioEventLoopGroup-2-1	3:27.259 (100.0%)	0.0 (0.0%)	0.0 (0.0%)	
VM JFR Buffer Thread	3:27.259 (100.0%)	0.0 (0.0%)	0.0 (0.0%)	
Attach Listener	3:27.259 (100.0%)	0.0 (0.0%)	0.0 (0.0%)	
Signal Dispatcher	3:27.259 (100.0%)	0.0 (0.0%)	0.0 (0.0%)	
RMI TCP Connection(17)-192.168.1.103	2:46.146 (80.1%)	0.0 (0.0%)	41.113 (19.9%)	
RMI TCP Connection(20)-192.168.1.103	2:44.200 (79.2%)	0.0 (0.0%)	43.059 (20.7%)	
RMI TCP Connection(22)-192.168.1.103	2:14.244 (65.6%)	0.0 (0.0%)	1:10.122 (34.3%)	
RMI TCP Connection(25)-192.168.1.103	44.042 (100.0%)	0.0 (0.0%)	0.0 (0.0%)	
JMX server connection timeout 34	7.979 (3.8%)	0.0 (0.0%)	3:19.280 (96.1%)	
RMI TCP Connection(14)-192.168.1.103	0.997 (0.4%)	0.0 (0.0%)	3:26.262 (99.5%)	
JMX server connection timeout 32	0.0 (0.0%)	0.0 (0.0%)	3:27.259 (100.0%)	
RMI Scheduler(0)	0.0 (0.0%)	0.0 (0.0%)	3:27.259 (100.0%)	
nioEventLoopGroup-3-1	0.0 (0.0%)	0.0 (0.0%)	3:27.259 (100.0%)	
JFR request timer	0.0 (0.0%)	0.0 (0.0%)	3:27.259 (100.0%)	
Finalizer	0.0 (0.0%)	0.0 (0.0%)	3:27.259 (100.0%)	

图 7-17 线程运行状态

Dump 线程堆栈，发现 Netty 的 NIO 线程被后端业务自定义的阻塞队列阻塞：

```
~nioEventLoopGroup-3-1" prio=10 tid=0x000000000bbd000 nid=0x44a48 waiting on condition [0x000000000cc8e000]
  java.lang.Thread.State: WAITING (parking)
    at sun.misc.Unsafe.park(Native Method)
    - parking to wait for <0x00000000c2e53188> (a java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject)
    at java.util.concurrent.locks.LockSupport.park(LockSupport.java:186)
    at java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.await(AbstractQueuedSynchronizer.java:2043)
    at java.util.concurrent.ArrayBlockingQueue.put(ArrayBlockingQueue.java:324)
    at io.netty.example.echo2.EchoServerHandler.mockBlock(EchoServerHandler.java:66)
    at io.netty.example.echo2.EchoServerHandler.channelRead(EchoServerHandler.java:57)
    at io.netty.channel.ChannelHandlerInvokerUtil.invokeChannelReadNow(ChannelHandlerInvokerUtil.java:74)
    at io.netty.channel.DefaultChannelHandlerInvoker.invokeChannelRead(DefaultChannelHandlerInvoker.java:138)
    at io.netty.channel.DefaultChannelHandlerContext.fireChannelRead(DefaultChannelHandlerContext.java:320)
    at io.netty.channel.DefaultChannelPipeline.fireChannelRead(DefaultChannelPipeline.java:846)
    at io.netty.channel.nio.AbstractNioByteChannel$NioByteUnsafe.read(AbstractNioByteChannel.java:127)
    at io.netty.channel.nio.NioEventLoop.processSelectedKey(NioEventLoop.java:485)
    at io.netty.channel.nio.NioEventLoop.processSelectedKeysOptimized(NioEventLoop.java:452)
    at io.netty.channel.nio.NioEventLoop.run(NioEventLoop.java:346)
    at io.netty.util.concurrent.SingleThreadEventExecutor$5.run(SingleThreadEventExecutor.java:795)
    at java.lang.Thread.run(Thread.java:745)
```

7.6.3 案例总结

在任何情况下，都不能阻塞 Netty 的 NIO 线程，建议如下：

1. 不要在 Netty 的 NIO 线程中做可能会导致阻塞的操作，例如同步磁盘 I/O 读

写、数据库访问、FTP访问等；

2. 消息由通信模块向业务逻辑传递时，往往使用线程池、消息队列等做线程切换，建议不要使用可能会导致同步阻塞的操作，例如阻塞队列的PUT方法。如果后端队列已满，可以使用动态流控、系统拥塞保护等机制，快速失败，而不是同步等待。

7.7 性能数据统计类

7.7.1 问题描述

某生产环境在业务高峰期，偶现服务调用时延突刺问题，时延突然增大的服务没有固定规律，比例虽然很低，但是对客户体验影响很大，需要尽快定位出问题原因并解决。

7.7.2 问题分析

服务调用时延增大，但并不是异常，因此运行日志并不会打印 ERROR 日志，单靠传统的日志无法进行有效问题定位。利用分布式消息跟踪系统魔镜，进行分布式环境的故障定界。

通过对服务调用时延进行排序和过滤，找出时延增大的服务调用链详细信息，发现业务服务端处理很快，但是消费者统计数据却显示服务端处理非常慢，调用链两端看到的数据不一致，怎么回事？

对调用链的埋点日志进行分析发现，服务端打印的时延是业务服务接口调用的时延，并没有包含：

- 通信端读取数据报、消息解码和内部消息投递、队列排队的时间；
- 通信端编码业务消息、在通信线程队列排队时间、消息发送到Socket的时间。

调用链的工作原理如图 7-18 所示。

将调用链中的消息调度过程详细展开，以服务端读取请求消息为例进行说明，如图 7-19 所示。

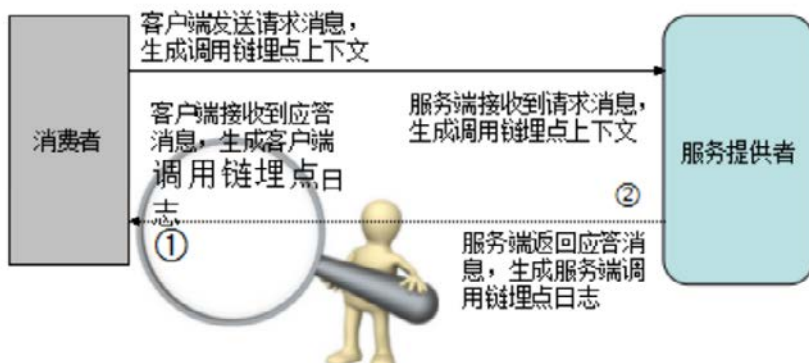


图 7-18 调用链工作原理

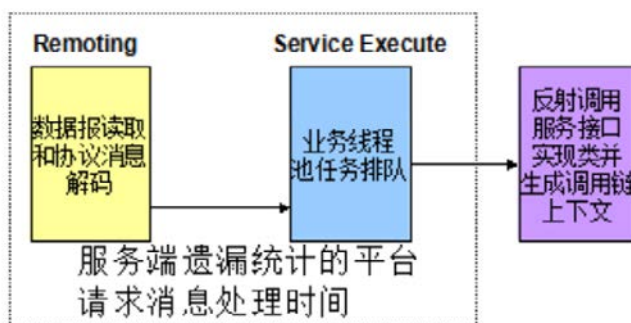


图 7-19 性能统计日志埋点

优化调用链埋点日志，措施如下：

- 包含客户端和服务端消息编码和解码的耗时
- 包含请求和应答消息在业务线程池队列中的排队时间；
- 包含请求和应答消息在通信线程发送队列（数组）中的排队时间

同时，为了方便问题定位，我们需要打印输出 Netty 的性能统计日志，主要包括：

- 每条链路接收的总字节数、周期T接收的字节数、消息接收CAPs
- 每条链路发送的总字节数、周期T发送的字节数、消息发送CAPs

优化之后，上线运行一天之后，我们通过分析比对 Netty 性能统计日志、调用链日志，发现双方的数据并不一致，Netty 性能统计日志统计到的数据与前端门户看到的也不一致，因为怀疑是新增的性能统计功能存在 BUG，继续问题定位。

首先对消息发送功能进行 CodeReview，发现代码调用完 writeAndFlush 之后直接对发送的请求消息字节数进行计数，代码如下：

```
int sendBytes = poolBuf.readableBytes();
ctx.writeAndFlush(poolBuf);
totalSendBytes += sendBytes;
```

实际上，调用 writeAndFlush 并不意味着消息已经发送到网络上，它的功能分解如下：

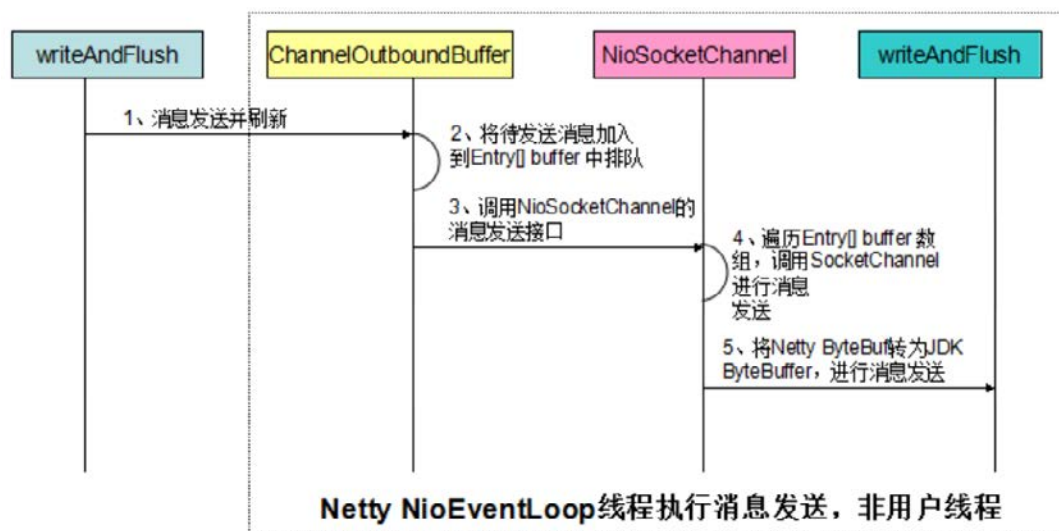


图 7-20 writeAndFlush 工作原理图

通过对 writeAndFlush 方法展开分析，我们发现性能统计代码存在如下几个问题：

- 业务ChannelHandler的执行时间
- ByteBuf在ChannelOutboundBuffer 数组中排队时间
- NioEventLoop线程调度时间，它不仅仅只处理消息发送，还负责数据报读取、定时任务执行以及业务定制的其它I/O任务
- JDK NIO类库将ByteBuffer写入到网络的时间，包括单条消息的多次写半包

由于性能统计遗漏了上述 4 个步骤的执行时间，因此统计出来的性能比实际

值更高，这会干扰我们的问题定位。

7.7.3 问题总结

其它常见性能统计误区汇总：

调用 write 方法之后就开始统计发送速率，示例代码如下：

```
int sendBytes = poolBuf.readableBytes();
ctx.write(poolBuf);
totalSendBytes+= sendBytes;
```

消息编码时进行性能统计，示例代码如下：

```
/**
 * Encode a message into a {@link ByteBuffer}. This method will be called for each written message
 * that can be handled
 * by this encoder.
 *
 * @param ctx the {@link ChannelHandlerContext} which this {@link MessageToByteEncoder}
 * @param msg the message to encode
 * @param out the {@link ByteBuffer} into which the encoded message will be written
 * @throws Exception is thrown if an error occurs
 */
protected abstract void encode(ChannelHandlerContext ctx, I msg, ByteBuffer out) throws Exception;
```

编码之后，获取 out 可读的字节数，然后做累加。编码完成，ByteBuffer 并没有被加入到发送队列（数组）中，因此在此时做性能统计仍然是不准的。

正确的做法：

- 调用writeAndFlush方法之后获取ChannelFuture;
- 新增消息发送ChannelFutureListener，监听消息发送结果，如果消息写入网络Socket成功，则Netty会回调ChannelFutureListener的operationComplete方法;
- 在消息发送ChannelFutureListener的operationComplete方法中进行性能统计。

示例代码如下：

```
final int sendBytes = poolBuf.readableBytes();
ChannelFuture writeFuture = ctx.writeAndFlush(poolBuf);
writeFuture.addListener(new ChannelFutureListener() {
    @Override
    public void operationComplete(ChannelFuture future) throws Exception {
        totalSendBytes+= sendBytes;
    }
});
```

问题定位出来之后，按照正确的做法对 Netty 性能统计代码进行了修正，上线之后，结合调用链日志，很快定位出了业务高峰期偶现的部分服务时延毛刺较大问题，优化业务线程池参数配置之后问题得到解决。

7.8 初始化启动类

7.8.1 问题描述

问题 1：当我启动 netty 服务器时使用阻塞方式绑定监听端口：

```
bootstrap.bind(PORT)

.sync().channel()

.closeFuture().sync();
```

假如我注释掉最有一行变成：

```
bootstrap.bind(PORT)

.sync().channel()
```

这样的话服务器进程启动后马上就返回退出了，不再继续监听。请问一下这其中的原理是什么？

问题 2：假如必须要加上 `.closeFuture().sync()` 这行代码，能不能采用下面监听器的方法进行监听套接字关闭事件：

```
f.channel().closeFuture().addListener(new ChannelFutureListener() {

    @Override

    public void operationComplete(ChannelFuture future)

throws Exception {

        // 业务逻辑处理代码，此处省略 ...

        log.info(future.channel().toString() + “ 链路关闭 ”);

    }

}); .
```

然而我自己测试了一下，这样去监听套接字的关闭事件，好像还是会出现服务器套接字直接关闭，进程退出的情况。请问这又是为什么呢？

7.8.2 答疑解惑

在回答上面两个问题之前，先普及下 Java Daemon 线程，这样你才能够把问题理解透彻。

所谓守护线程（Daemon）就是运行在程序后台的线程，程序的主线程 Main 不是守护线程。Daemon Thread 在 Java 里面的定义是，如果虚拟机中只有 Daemon thread 运行，则虚拟机退出。

1. 虚拟机中可能会同时有多个线程在运行，只有当所有的非守护线程都结束的时候，虚拟机的进程才会结束，不管在运行的线程是不是main()线程。
2. Main主线程结束了，如果此时运行的其它Thread全部是Daemon thread，JVM会使这些Threads停止，同时JVM退出。如果此时正在运行的其它线程有非守护线程，那么必须等所有的非守护线程结束之后，JVM才会退出。

举例如下：

```
public static void main(String[] args) throws
IllegalArgumentException, InterruptedException {
    System.out.println("Start time is : " + new Date());
    Thread t = new Thread(new Runnable() {
        public void run() {
            try {
                TimeUnit.DAYS.sleep(Long.MAX_VALUE);
            } catch (InterruptedException e) {
                e.printStackTrace();}}
        }, "Daemon-T");
    //t.setDaemon(false);
    t.setDaemon(true);
}
```

```

t.start();

TimeUnit.SECONDS.sleep(15);

System.out.println("End time is : " + new Date());}}

```

执行结果：因为是 Daemon 线程，当主线程执行 15S 退出之后，进程退出：

```

Start time is : Sat Apr 30 11:41:22 CST 2016
End time is : Sat Apr 30 11:41:37 CST 2016

```

修改上述代码：

```

public static void main(String[] args) throws
IllegalArgumentException, InterruptedException {

    System.out.println("Start time is : " + new Date());
    Thread t = new Thread(new Runnable() {
        public void run() {
            try {
                TimeUnit.DAYS.sleep(Long.MAX_VALUE);
            } catch (InterruptedException e) {
                e.printStackTrace();}}
        }, "Daemon-T");

    t.setDaemon(false);

    //    t.setDaemon(true);

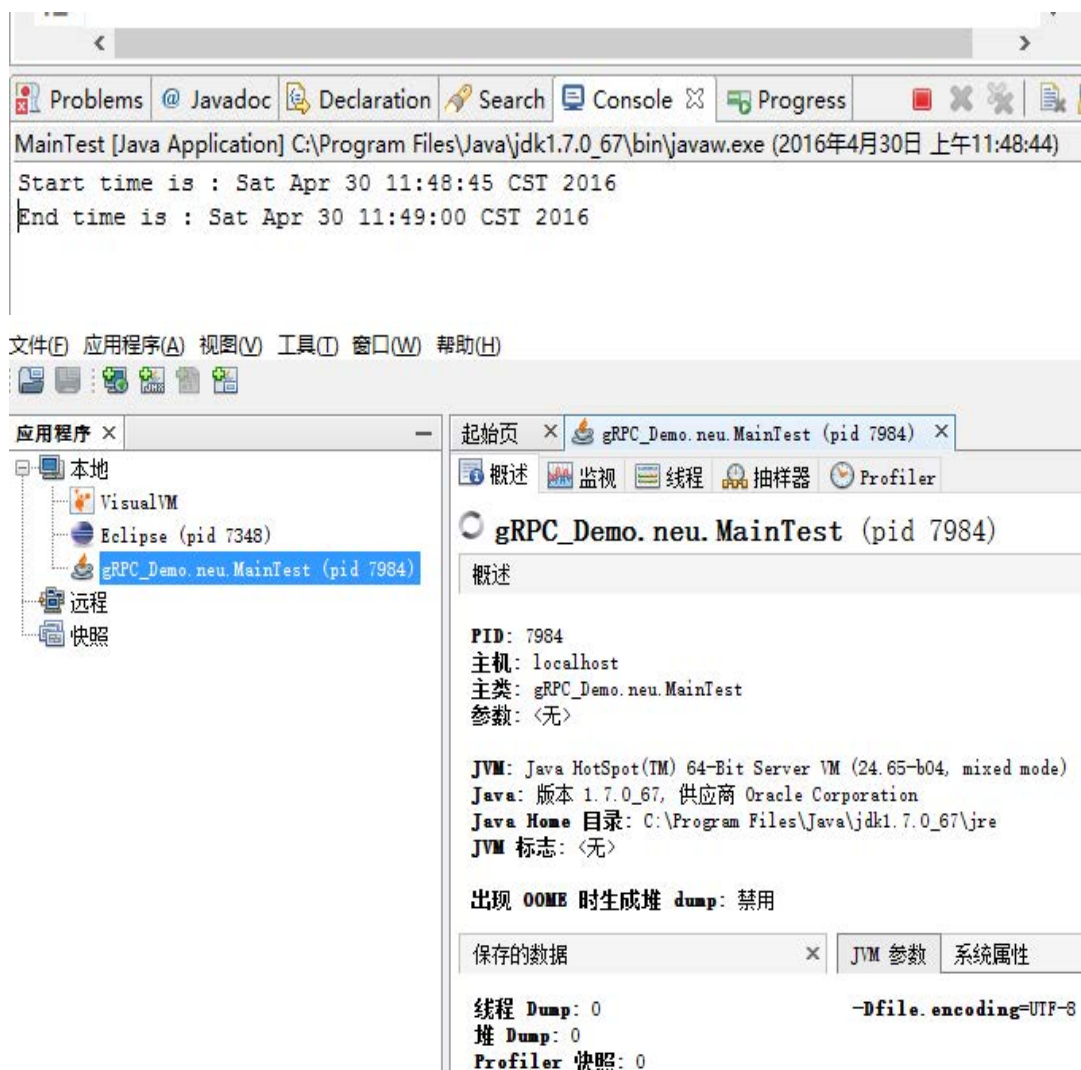
    t.start();

    TimeUnit.SECONDS.sleep(15);

    System.out.println("End time is : " + new Date());}}

```

执行结果：因为存在非 Daemon 线程执行未结束，及时主线程执行 15S 结束之后，JVM 进程仍然不会退出。



然后我们来回答之前的问题:

`bootstrap.bind(PORT)`

`.sync().channel()`

在 Netty 中, 绑定端口的操作并不是在主线程中进行的, 真正执行它的是 NioEventLoop 线程, 调试堆栈如下图所示。

它最终的执行结果其实就是调用了 Java NIO Socket 的绑定操作, 如下所示。

`javaChannel().socket().bind(localAddress, config.getBacklog());`

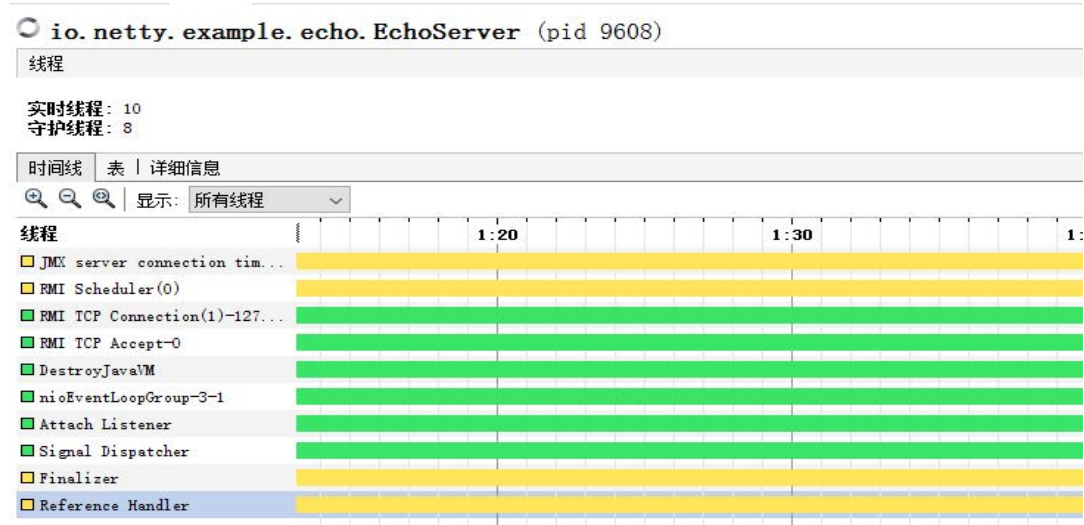
绑定操作执行完成之后, main 函数主线程就不会阻塞, 如果后续没有代码, main 线程就会退出, main 线程退出是否意味着 JVM 进程退出, 非也。

```

DefaultChannelPipeline$HeadHandler.bind(ChannelHandlerContext, SocketAddress, ChannelPromi
ChannelHandlerInvokerUtil.invokeBindNow(ChannelHandlerContext, SocketAddress, ChannelProi
DefaultChannelHandlerInvoker.invokeBind(ChannelHandlerContext, SocketAddress, ChannelProm
DefaultChannelHandlerContext.bind(SocketAddress, ChannelPromise) line: 366
LoggingHandler.bind(ChannelHandlerContext, SocketAddress, ChannelPromise) line: 249
ChannelHandlerInvokerUtil.invokeBindNow(ChannelHandlerContext, SocketAddress, ChannelProi
DefaultChannelHandlerInvoker.invokeBind(ChannelHandlerContext, SocketAddress, ChannelProm
DefaultChannelHandlerContext.bind(SocketAddress, ChannelPromise) line: 366
DefaultChannelPipeline.bind(SocketAddress, ChannelPromise) line: 898
NioServerSocketChannel(AbstractChannel).bind(SocketAddress, ChannelPromise) line: 189
AbstractBootstrap$2.run() line: 309
NioEventLoop(SingleThreadEventExecutor).runAllTasks(long) line: 319
NioEventLoop.run() line: 355

```

之前已经讲过，待所有非守护线程全部执行完成之后，进程才会退出。那此时是否还存在其它非守护线程运行吗，我们首先看下线程状态：



我们发现，Main 主线程已经运行结束，但是 Netty 的 NioEventLoop 还处于运行状态，因此 JVM 进程并没有退出。

通过对 NioEventLoop 源码分析（此处略），我们发现：

1. NioEventLoop 是非守护线程；
2. NioEventLoop 运行之后，不会主动退出；
3. 只有调用 `shutdownGracefully` 方法，NioEventLoop 才会主动退出。

按照这个分析，及时 `main()` 函数执行结束，进程也不会退出啊，为啥注释

掉 `.closeFuture().sync()`；方法之后，进程退出了呢？

这个问题很简单，因为 Netty 官方的 Demo 中，标准的代码都是这样写的：

```
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();

}
```

如果注释掉 `f.channel().closeFuture().sync()`；则端口绑定成功之后，`Main()` 函数退出之前会执行 `bossGroup.shutdownGracefully()`；它会使所有的 `NioEventLoop` 优雅退出，导致所有非守护进程执行结束，最终 JVM 进程退出。

假如我们把代码修改成这样：

```
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();

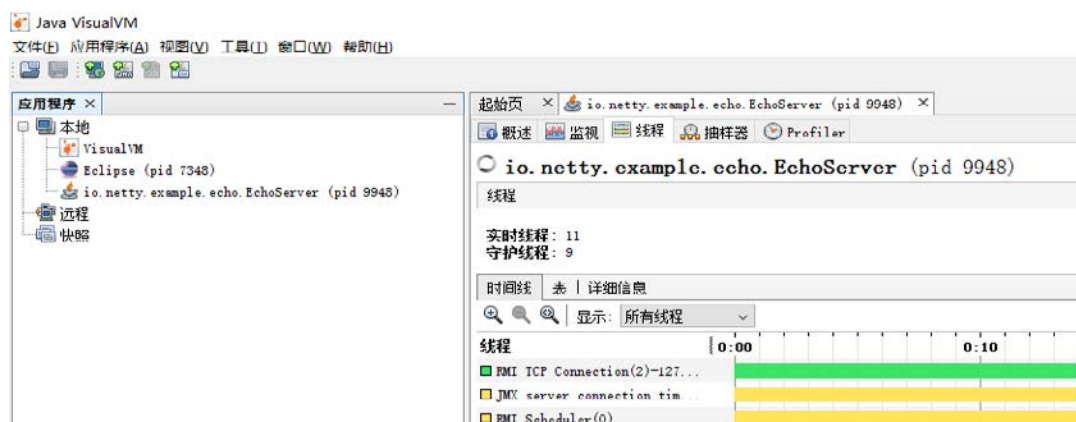
}
```

就会发现 JVM 进程不再退出。

接着回答第二个问题，把代码做了修改，不调用 `.closeFuture().sync()`，而是改用监听器尝试监听链路关闭事件。

监听器实际是一种异步回调方式，如果你不想通过同步阻塞的方式等待结果，

而是希望程序继续往下走，当执行结束之后通过系统回调的方式通知你，由你来实现自己的逻辑。



通过上面的分析可以得出这样的结论：

- 改用监听器监听链路关闭事件，不会阻塞Main()线程；
- 端口绑定成功之后，Main线程继续向下执行；
- 由于你在finnal中增加了线程池关闭代码，所以NioEventLoop线程主动退出；
- 系统没有正在运行的非守护线程，JVM进程退出。

7.8.3 问题总结

这个案例非常典型，有很多初学者都咨询过类似的问题。从问题本身看，有几个知识点必须要掌握，才能够更灵活的使用 Netty：

1. Java Deamon线程工作机制；
2. Netty的NioEventLoop线程工作原理；
3. Netty NIO监听器的使用方式以及原理。

作者简介

李林锋，2007年毕业于东北大学，2008年进入华为公司从事高性能通信软件的设计和开发工作，有7年NIO设计和开发经验，精通Netty、Mina等NIO框架和平台中间件，现任华为软件平台开放实验室架构师，《Netty权威指南》、《分布式服务框架原理与实践》作者。

联系方式：新浪微博 Nettying 微信：Nettying 微信公众号：Netty之家

对于Netty学习中遇到的问题，或者认为有价值的Netty或者NIO相关案例，可以通过上述几种方式联系我。

版权声明

InfoQ 中文站出品

架构师特刊：深入浅出 Netty

©2016 极客邦控股（北京）有限公司

本书版权为极客邦控股（北京）有限公司所有，未经出版者预先的书面许可，不得以任何方式复制或抄袭本书的任何部分，本书任何部分不得用于再印刷，存储于可重复使用的系统，或者以任何方式进行电子、机械、复印和录制等形式传播。

本书提到的公司产品或者使用到的商标为产品公司所有。

如果读者要了解具体的商标和注册信息，应该联系相应的公司。

出版：极客邦控股（北京）有限公司

北京市朝阳区洛娃大厦 C 座 1607

欢迎共同参与 InfoQ 中文站的内容建设工作，包括原创投稿和翻译，请联系 editors@cn.infoq.com。

网 址：www.infoq.com.cn