



Netty

跟闪电侠学Netty

Netty即时聊天实战与底层原理

闪电侠（俞超）著

闪电侠



中国工信出版集团



电子工业出版社
<http://www.phei.com.cn>

Broadview
www.broadview.com.cn



Netty

跟闪电侠学Netty

Netty即时聊天实战与底层原理

闪电侠（俞超）著

闪电侠



中国工信出版集团



电子工业出版社

PUBLISHING HOUSE OF ELECTRONIC INDUSTRY

<http://www.phei.com.cn>

版权信息

COPYRIGHT

书名：跟闪电侠学Netty：Netty即时聊天实战与底层原理

作者：俞超

出版社：电子工业出版社

出版时间：2022年3月

ISBN：9787121426797

字数：248千字

本书由电子工业出版社有限公司授权得到APP电子版制作与发行

版权所有·侵权必究

内容简介

这是一本专门为Netty初学者打造的入门及进阶学习图书，无论你之前有没有使用过Netty，都可以从本书中有所收获。

本书分上下两篇。上篇通过一个即时聊天系统的实战案例，让读者能够系统地使用一遍Netty，全面掌握Netty的知识点；下篇通过对源码的层层剖析，让读者能够掌握Netty底层原理，知其然并知其所以然，从而编写出高性能网络应用程序。

如果你想全面系统地学习Netty，并掌握一些性能调优方法，本书上篇可以帮助你完成这个目标。如果你想深入了解Netty的底层设计，编写出更灵活高效的网络通信程序，本书下篇可以帮助你完成这个目标。如果你从未读过开源框架源码，本书将是你的第一本源码指导书，读源码并不难，难的是迈出这一小步，之后就能通往更广阔的世界。

推荐语

Netty是业界卓越的开源大作！第一次使用时，我就被它的高吞吐和高性能所折服。JBoss开源的Netty网络框架是CAT服务端除JDK外唯一的外部依赖，表现不俗，生产环境物理机上几张千兆网卡经常可以满负荷运转，多核CPU可以保持100%运行，让CAT的单机吞吐能力最高达到45万MPS，消息平均大小约1KB。我强烈建议大家使用Netty，当然我们也踩过几个坑，需要合理使用，希望本书对大家理解和使用Netty网络编程有所帮助。

——平安银行零售首席架构师 吴其敏

曾几何时，Mina很火，然而当Netty横空出世的时候，却抢尽了风头。作为高性能领域的杰出代表，Netty成为很多RPC框架底层通信的标配，闪电侠的这本书从典型的通信场景——聊天室出发，深入浅出地剖析了Netty的源码，对于想写出高性能代码的读者来说，非常值得借鉴，值得一读。

——HeapDump性能社区发起人 & PerfMa CEO你假笨

闪电侠是个精力充沛的技术大牛，花了大量的业余时间投身于开源代码和撰写技术博客。在美团工作期间，我有幸与闪电侠合作多年，一起搭建了美团重要的网络基础设施，并凭借该项目一举拿下了美团的年度最高技术奖项。在分工合作上，我负责前端，闪电侠负责后端。他当时对Netty技术非常痴迷，将通信框架进行大量重构，使得性能获得极大提升，此外还进行了大量的技术创新，来保障通信高性能和系统稳定运行。更难能可贵的是，他还撰写了很多关于Netty的研究心得文章。现在，这些宝贵经验已经集结成书，强烈建议所有对Netty技术或高性能网络技术感兴趣的技术人员阅读。

——平安银行前端架构领域负责人 周辉

在异步编程模型上，Netty无疑是值得学习和研究的NIO框架。作者从原理和案例实战的角度出发，全面介绍了Netty的使用。本书条理清晰，对技术的阐述循序渐进，囊括了Netty所涉及的NIO、Selector模型、线程模型、网络协议等诸多核心技术，强烈建议大家阅读学习。

——《高可用可伸缩微服务架构》作者 程超

本书沉淀了笔者自身多年研发实践和进阶心得，通过精巧设计的案例，将网络通信、并发编程的重点知识与具体场景结合起来，让读者快速入门，掌握生产级别的Netty开发技能。与此同时，完整、系统化的源码分析，也能够使工程师有效提升底

层技术能力，形成良好的工程素养。相信这些一手修炼心得会让开发者受益匪浅，非常适合想要入门或者进阶Netty开发的初、中级工程师。

——Oracle前首席工程师 杨晓峰

推荐序

Netty是一个互联网公司非常流行的Java开源网络应用程序框架，可以帮助用户快速开发高性能、高稳定的网络通信服务。作为最有影响力的NIO框架，Netty得到了众多架构师和程序员的喜爱，并且在互联网基础架构、游戏行业、大数据通信等领域都有广泛的应用。尤其在互联网基础架构方面，Netty几乎一统江湖。我有幸参与美团点评基础架构快速发展的十年，Netty几乎在所有的核心架构中间件都有广泛使用，包括美团点评监控系统Cat、美团点评移动端长连网关Shark、美团点评RPC服务框架Pigeon及Octo等。

Netty之所以成为Java通信框架的主要核心组件，我总结主要是因为以下3点：

- 1.设计优雅：Netty封装了Java中各类NIO通信的Channel，定义了统一数据传输的ByteBuf，让开发更加简单，容易上手使用。
- 2.高性能：Netty做了大量的底层技术优化，包括Reactor线程模型、Zero-Copy技术，减少了资源消耗，在一个4核8G内存的机器上进行压测，性能可以达到10万QPS。
- 3.社区氛围，持续运营维护，有持续更新的文档、指南等，用户遇到的问题几乎都能在社区找到解答。

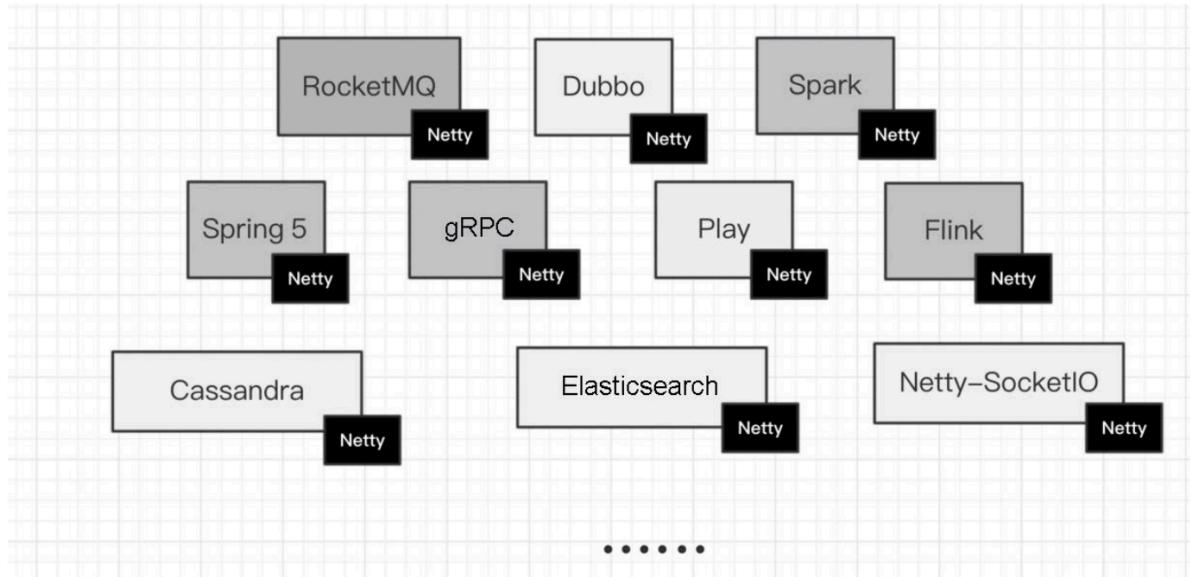
本书作者闪电侠同学，多年在研发一线，在平时的工作中对Netty的钻研非常深入，在公司内作为核心技术骨干，不仅仅解决本团队的技术难题，也帮助其他团队解决相关技术疑难杂症。很荣幸能成为最早的阅读者之一，这本书深入浅出、言简意赅，通过一个通信程序的案例介绍了Netty的基础知识，并通过源码分析介绍了Netty的底层原理，希望给各位读者带来帮助。

——美团前技术总监 & 比心技术CTO尤勇

前言

为什么写这本书

Netty是互联网中间件、大数据领域使用最广泛、最核心的网络通信框架。由下图可以看到，几乎所有互联网中间件、大数据框架均使用了Netty，掌握Netty是一名初、中级工程师迈向中、高级工程师所需的最重要的技能之一。



然而笔者发现，目前市面上对初学者友好的Netty资料较少，网络上大多数技术博客都是一堆零散的知识点集合，无法串成一条线、形成一个体系。笔者在多年的Netty实战、调优、“踩坑”过程中积累了丰富的经验，本书中笔者将这些经验系统地分享给大家，帮助大家提升核心竞争力。

本书上篇通过一个即时聊天系统的实战案例，让读者能够系统地使用一遍Netty，全面掌握Netty的知识点；下篇通过对源码的层层剖析，让读者能够掌握Netty底层原理，知其然并知其所以然，从而编写出高性能网络应用程序。

上篇 入门实战

在入门实战篇中，读者跟随笔者实践完这个即时聊天系统后，能够学会如何使用Netty完成最基本的网络通信程序，可以掌握以下知识点。

- 1.如何启动服务端？

- 2.如何启动客户端？
- 3.如何设计长连自定义协议？
- 4.拆包/粘包原理与实践。
- 5.如何实现自定义编解码？
- 6.如何使用Pipeline与ChannelHandler？
- 7.心跳与空闲检测的方法。
- 8.如何进行性能调优？

下篇 源码分析

在源码分析篇中，笔者从用户视角出发，环环相扣，带领读者逐个攻破Netty底层原理，掌握以下知识点。

- 1.服务端启动流程：ServerBootstrap外观，创建NioServerSocketChannel，初始化，注册Selector，绑定端口，接收新连接。
- 2.高并发线程模型：Netty无锁化串行设计，精心设计的Reactor线程模型榨干CPU、打满网卡、让应用程序性能爆表的底层原理。
- 3.新连接接入流程：boss Reactor线程，监测新连接，创建NioSocketChannel，IO线程分配，Selector注册事件。
- 4.解码原理：解码顶层抽象，定长解码器，行解码器，分隔符解码器，基于长度域解码器全面分析。
- 5.事件传播机制脉络：大动脉Pipeline，处理器ChannelHandler，Inbound和Outbound事件传播与异常传播的原理，编码原理。
- 6.writeAndFlush流程：深入了解使用最频繁的writeAndFlush的底层原理，避免踩坑。

适合人群

本书适合以下三类人群：

1.如果你听说过或简单使用过Netty，想全面系统地学习Netty，并掌握一些性能调优方法，本书的入门实战篇可以帮助你完成这个目标。

2.如果你深度使用过Netty，想深入了解Netty的底层设计，编写出更灵活高效的网络通信程序，本书的源码分析篇可以帮助你完成这个目标。

3.如果你从未读过开源框架源码，本书将是你的第一本源码指导书，阅读优秀的开源软件源码可以助你写出更优美的程序。读源码并不难，难的是迈出这一小步，之后就能通往更广阔的世界。

本书推荐使用方式

1.按章节顺序把入门实战篇的代码一章章输出到IDE中，在没有掌握前一章节的知识点之前，建议不要跳跃学习。

2.入门实战篇学完之后，合上书本，把本书即时聊天系统的代码再整体输出若干遍，输出的过程中可能会发现自己有遗忘知识点，这个时候可能需要不断翻阅书本，没关系，翻阅就好了。

3.确保最后一次实现本书的即时聊天系统的例子是没有翻阅书本的，是完全自行实现的，之后进入源码分析篇的学习。

4.针对源码分析篇，建议读者按章节顺序来学习，不要跳跃，不要图快，每一步都要扎实。

5.在源码学习的过程中，先跟随书本，对照源码（Netty 4.1.6），把对应章节的流程过一遍，每个章节学完之后，建议花较多的时间进行调试和阅读，确保掌握了前一章节的内容之后再进行下一章节的学习。

读者服务

微信扫码回复：42679



- 获取Netty进阶学习资料

- 加入本书读者交流群，与本书作者互动交流
- 获取【百场业界大咖直播合集】（持续更新），仅需1元

上篇

入门实战

◆

第1章 即时聊天系统简介 ◆

第2章 Netty是什么 ◆

第3章 Netty开发环境配置 ◆

第4章 服务端启动流程 ◆

第5章 客户端启动流程 ◆

第6章 客户端与服务端双向通信 ◆

第7章 数据载体ByteBuf的介绍 ◆

第8章 客户端与服务端通信协议编解码 ◆

第9章 实现客户端登录 ◆

第10章 实现客户端与服务端收发消息 ◆

第11章 Pipeline与ChannelHandler ◆

第12章 构建客户端与服务端的Pipeline ◆

第13章 拆包/粘包理论与解决方案 ◆

第14章 ChannelHandler的生命周期 ◆

第15章 使用ChannelHandler的热插拔实现客户端身份校验 ◆

第16章 客户端互聊的原理与实现 ◆

第17章 群聊的发起与通知 ◆

第18章 群聊的成员管理 ◆

第19章 群聊消息的收发及Netty性能优化 ◆

第20章 心跳与空闲检测

第1章

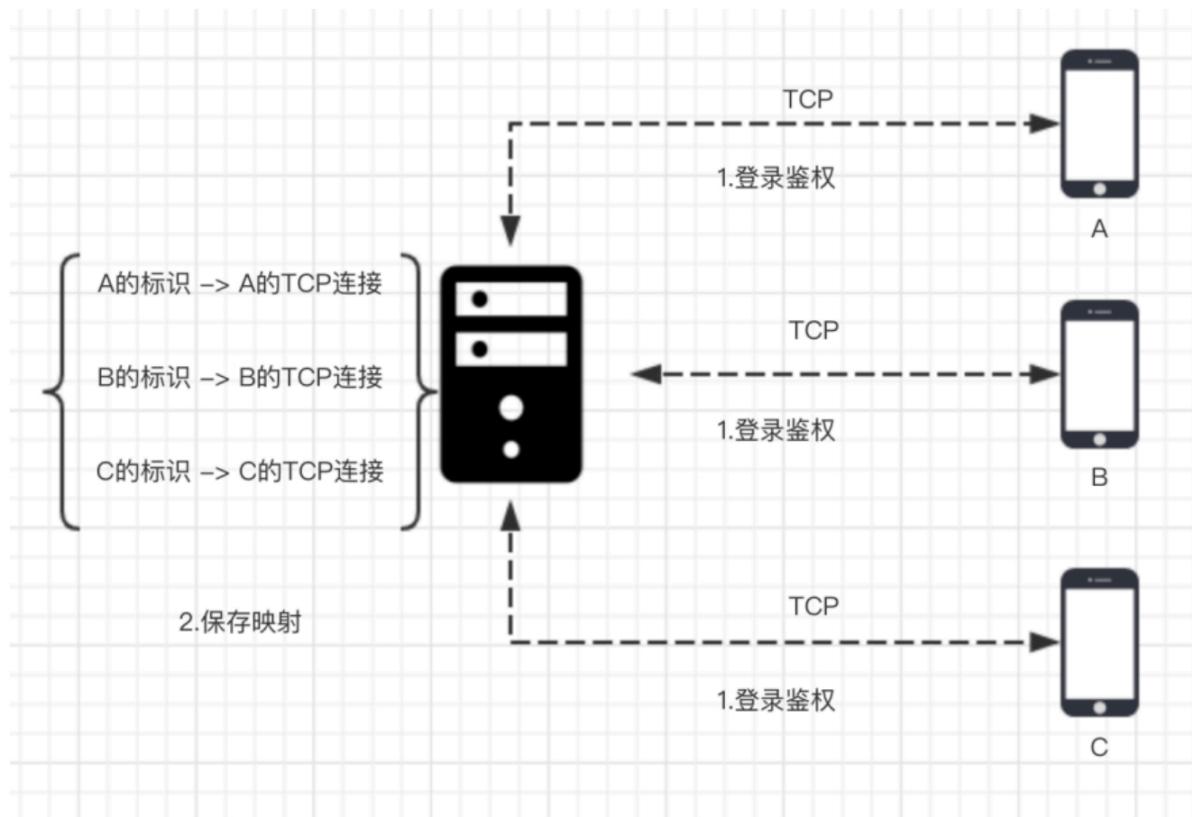
即时聊天系统简介

移动互联网时代，相信大家应该都对即时聊天工具不陌生，比如最常用的微信，从2011年1月21日诞生至今，已经成为国内数亿用户必不可少的即时通信工具，是男女老少手机中必备的顶级App。Netty是一个异步基于事件驱动的高性能网络通信框架，在互联网中间件领域网络通信层是无可争议的最强王者。在本书中，笔者将带领大家使用Netty一步一步实现即时聊天工具的核心功能。

即时聊天通常分为单聊和群聊，下面分别来介绍一下。

1.1 单聊流程

单聊指两个用户之间相互聊天。用户单聊的基本流程如下图所示。



1.A要和B聊天，首先A和B需要与服务端建立连接，然后进入登录流程，服务端保存用户标识和TCP连接的映射关系。

2.A给B发消息，首先需要将带有B标识的消息数据包发送到服务端，然后服务端从消息数据包中获得B的标识，找到对应B的连接，将消息发送给B。

3.任意一方发消息给对方，如果对方不在线，则需要将消息缓存，在对方上线之后再发送。

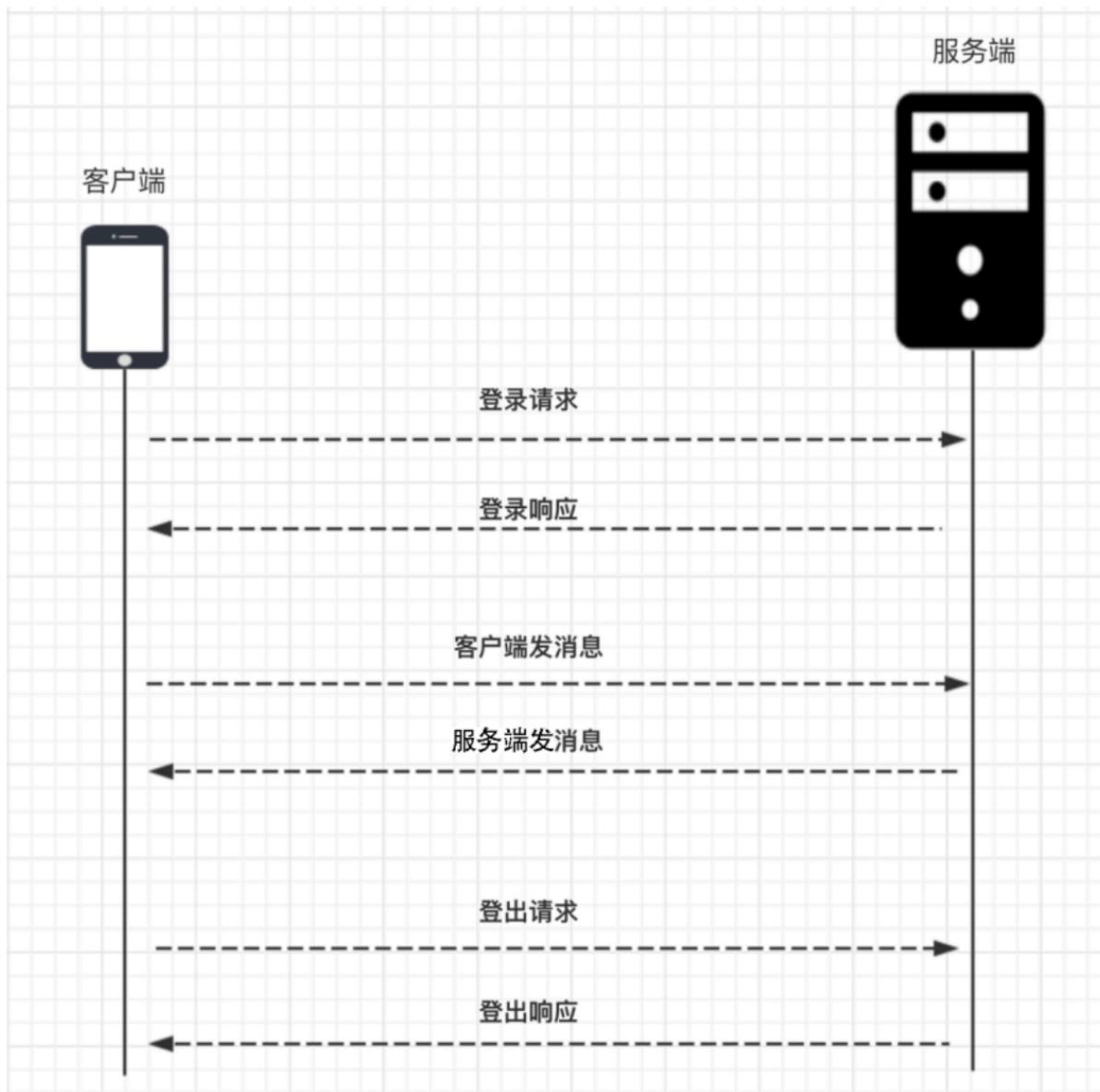
客户端与服务端之间相互通信的数据包被称为**指令数据包**。指令数据包分为指令和数据，每一种指令都对应客户端或者服务端的一种操作，数据部分对应的是指令处理需要的数据。

要实现单聊，客户端与服务端分别要实现哪些指令呢？下一节我们将详细介绍。

1.2 单聊的指令

1.2.1 指令图示

下图是客户端与服务端单聊的指令流程图。



1.2.2 指令列表

下表是本书中要实现的单聊的指令列表，每条指令都会分为客户端和服务端。

指令内容 客户端 服务端

登录请求 发送接收

登录响应 接收发送

客户端发消息 发送接收

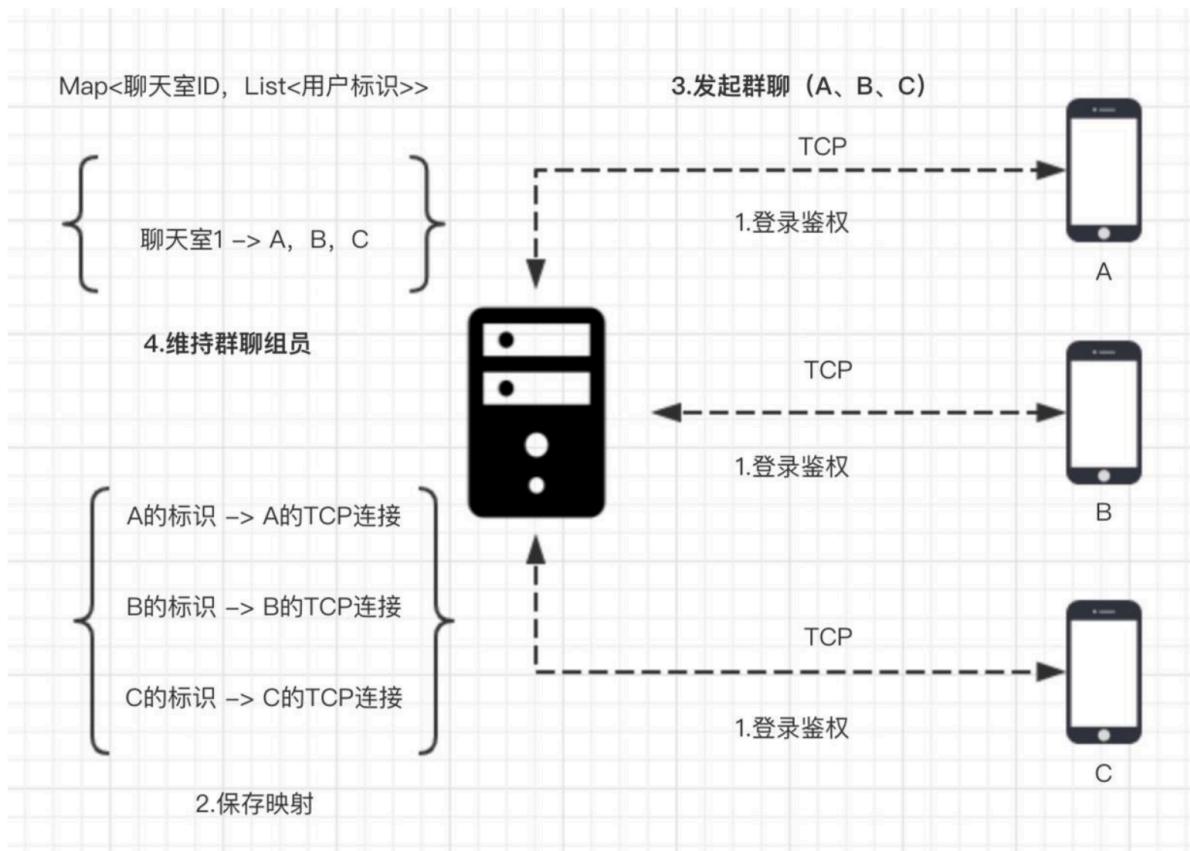
服务端发消息 接收发送

登出请求发送接收

登出响应接收发送

1.3 群聊流程

群聊指一个组内多个用户之间的聊天，一个用户发到群组的消息会被组内任何一个成员接收，群聊的基本流程如下图所示。



要实现群聊，其实流程和单聊类似。

1.A、B、C依然会经历登录流程，服务端保存用户标识对应的TCP连接。

2.A发起群聊的时候，将A、B、C的标识发送至服务端，服务端拿到标识之后建立一个群ID，然后把这个ID与A、B、C的标识绑定。

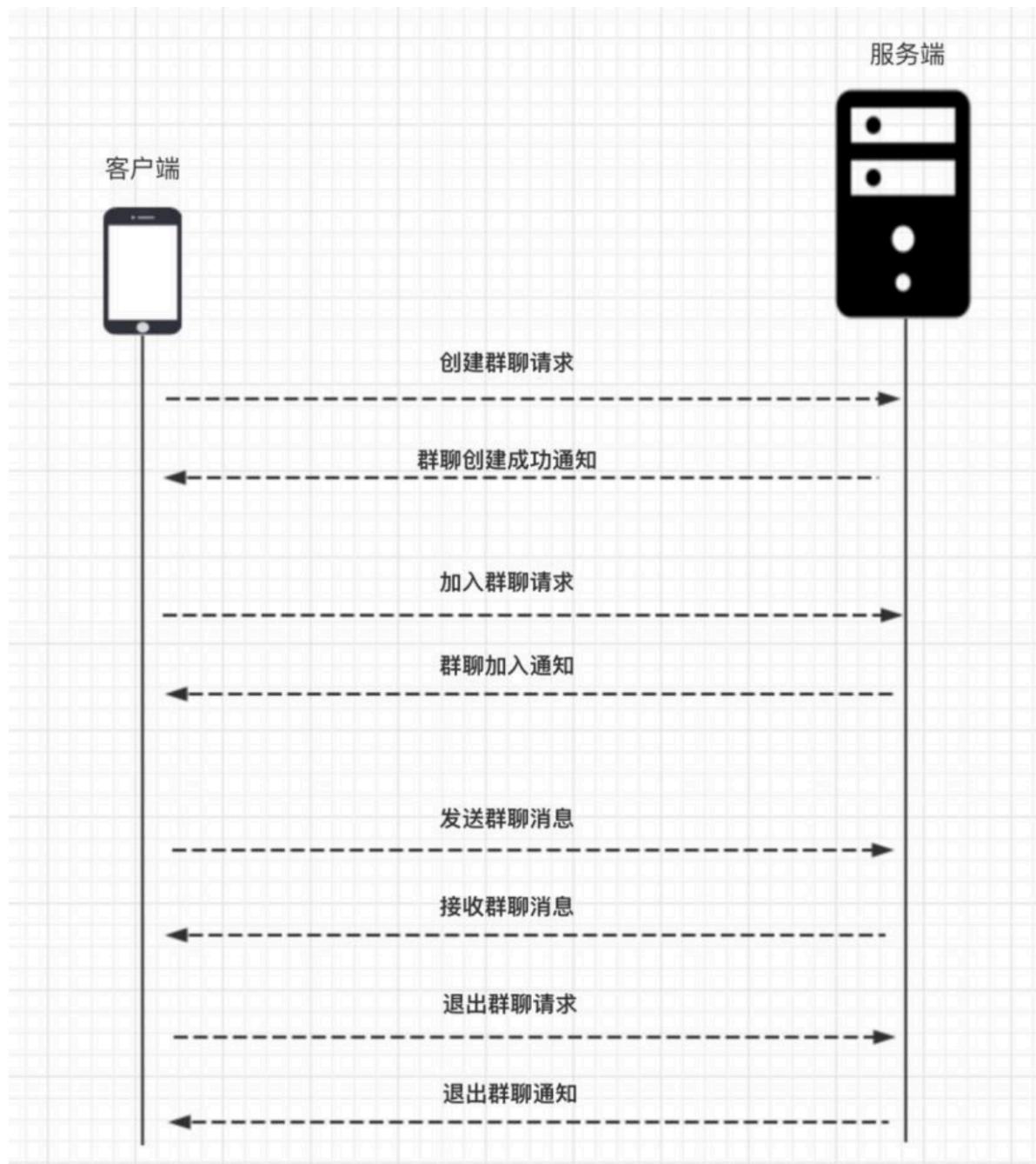
3.群聊中任意一方在群里聊天的时候，将群ID发送至服务端，服务端获得群ID之后，取出对应的用户标识，遍历用户标识对应的TCP连接，就可以将消息发送至每一个群聊成员。

群聊除了需要实现上述指令，还需要实现哪些指令呢？下一节将详细介绍。

1.4 群聊要实现的指令集

1.4.1 指令图示

群聊的指令图示如下图所示。



1.4.2 指令列表

群聊的指令如下表所示。

指令内容客户端服务端

创建群聊请求发送接收

群聊创建成功通知接收发送

加入群聊请求发送接收

群聊加入通知接收发送

发送群聊消息发送接收

接收群聊消息接收发送

退出群聊请求发送接收

退出群聊通知接收发送

1.5 Netty

本书统一使用IO表示I/O（Input/Output的缩写）。

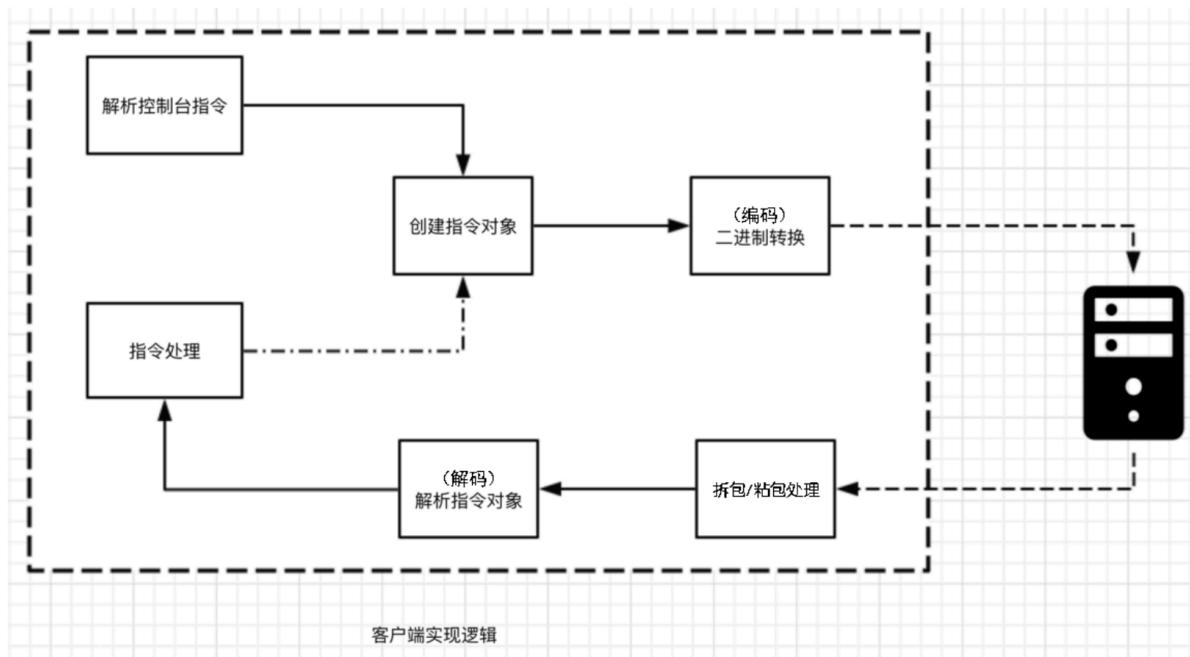
本书使用Netty统一的IO 读写API以及强大的Pipeline来编写业务处理逻辑，在后续章节中，会通过即时聊天系统这个例子，带大家逐步了解Netty以下核心知识点。

- 如何启动服务端？
- 如何启动客户端？
- 数据载体ByteBuf。
- 如何设计长连自定义协议？
- 拆包/粘包原理与实践。
- 如何实现自定义编解码？
- 如何使用Pipeline与ChannelHandler？
- 如何定时发心跳数据包？

- 如何进行连接空闲检测？

1.5.1 客户端使用Netty的程序逻辑结构

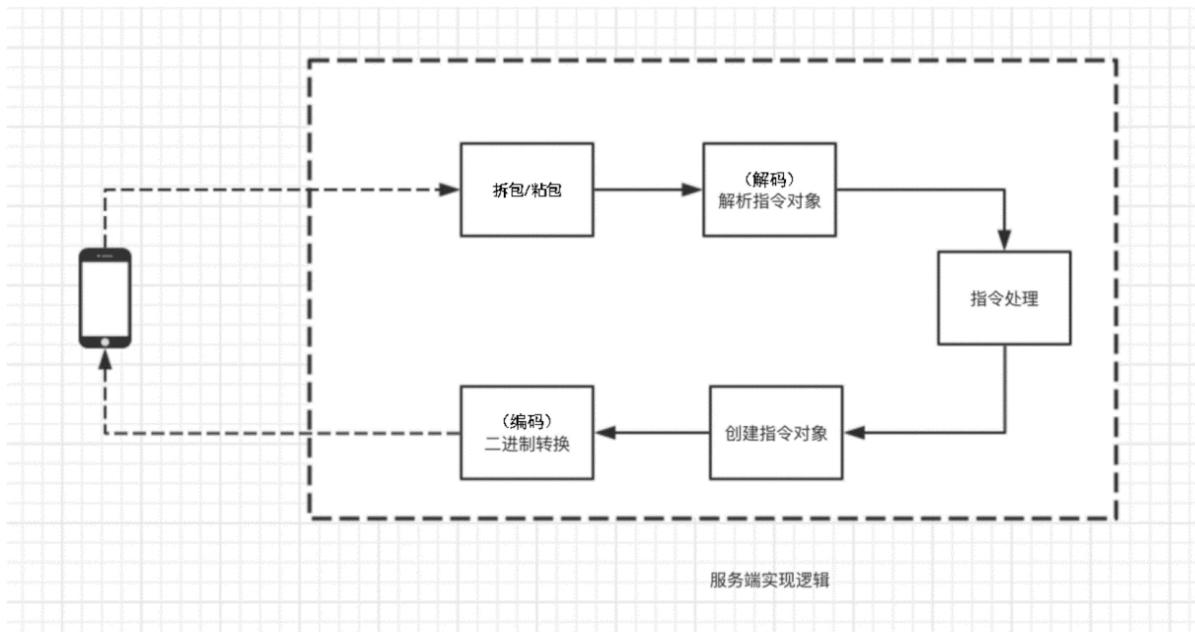
下图展示了客户端使用Netty的程序逻辑结构。



1. 客户端会解析控制台指令，比如发送消息或者建立群聊等指令。
2. 客户端会基于控制台的输入创建一个指令对象，用户告诉服务端具体要干什么事情。
3. TCP通信需要的数据格式为二进制，因此，接下来通过自定义二进制协议将指令对象封装成二进制，这一步被称为协议的编码。
4. 对于收到服务端的数据，首先需要截取出一段完整的二进制数据包（拆包/粘包相关的内容后续小节会讲解）。
5. 将此二进制数据包解析成指令对象，比如收到消息。
6. 将指令对象送到对应的逻辑处理器来处理。

1.5.2 服务端使用Netty的程序逻辑结构

服务端使用Netty的程序逻辑结构与客户端非常类似，如下图所示，这里不再赘述。



1.6 本书实现的即时聊天形式

由于本书以讲授Netty基础知识为主，故不会涉及即时聊天相关的图形化界面，后续所有的聊天都基于控制台进行，通过与控制台交互可以实现单聊和群聊。

第2章

Netty是什么

在开始了解Netty是什么之前，我们先来回顾一下，如果需要实现一个客户端与服务端通信的程序，使用传统的IO编程，应该如何来实现？

2.1 IO编程

我们简化一下场景：客户端每隔两秒发送一个带有时间戳的“hello world”给服务端，服务端收到之后打印它。

为了方便演示，在下面的例子中，服务端和客户端各有一个类，把这两个类复制到你的IDE中，先后运行IOServer.java和IOClient.java，可以看到效果。

下面是传统的IO编程中的服务端实现。

IOServer.java

```
/**  
 * @author 闪电侠  
 */  
  
public class IOServer {  
  
    public static void main(String [] args) throws Exception {  
  
        ServerSocket serverSocket = new ServerSocket(8000);  
  
        // 接收新连接线程  
  
        new Thread(() -> {  
  
            while (true) {  
  
                try {  
  
                    // (1)阻塞方法获取新连接  
                }  
            }  
        }).start();  
    }  
}
```

服务端首先创建一个serverSocket来监听8000端口，然后创建一个线程，线程里不断调用阻塞方法serverSocket.accept()获取新连接，见（1）；当获得新连接之后，为每

一个新连接都创建一个新线程，这个线程负责从该连接中读取数据，见（2）；然后以字节流方式读取数据，见（3）。

下面是传统的IO编程中的客户端实现。

IOClient.java

```
/**  
 * @author 闪电侠  
 */  
  
public class IOClient {  
  
    public static void main(String [] args) {  
  
        new Thread(() -> {  
  
            try {  
  
                Socket socket = new Socket("127.0.0.1", 8000);  
  
                while (true) {  
  
                    try {  
  
                        socket.getOutputStream().write((new Date() + ": hello  
world").getBytes());  
  
                        Thread.sleep(2000);  
  
                    } catch (Exception e) {  
  
                    }  
                }  
            } catch (IOException e) {  
  
            }  
        }  
    }  
}
```

```
    } ).start();  
  
    }  
  
}
```

客户端的代码相对简单，连接上服务端8000端口之后，每隔两秒，我们都向服务端写一个带有时间戳的“hello world”。

IO编程模型在客户端较少的情况下运行良好，但是对于客户端比较多的业务来说，单机服务端可能需要支撑成千上万个连接，IO模型可能就不太合适了，我们来分析一下原因。

在上面的示例中，从服务端代码可以看到，在传统的IO模型中，每个连接创建成功之后都需要由一个线程来维护，每个线程都包含一个while死循环，那么1万个连接对应1万个线程，继而有1万个while死循环，这就带来如下几个问题。

- 1.线程资源受限：线程是操作系统中非常宝贵的资源，同一时刻有大量的线程处于阻塞状态，是非常严重的资源浪费，操作系统耗不起。
- 2.线程切换效率低下：单机CPU核数固定，线程爆炸之后操作系统频繁进行线程切换，应用性能急剧下降。
- 3.除了以上两个问题，在IO编程中，我们看到数据读写是以字节流为单位的。

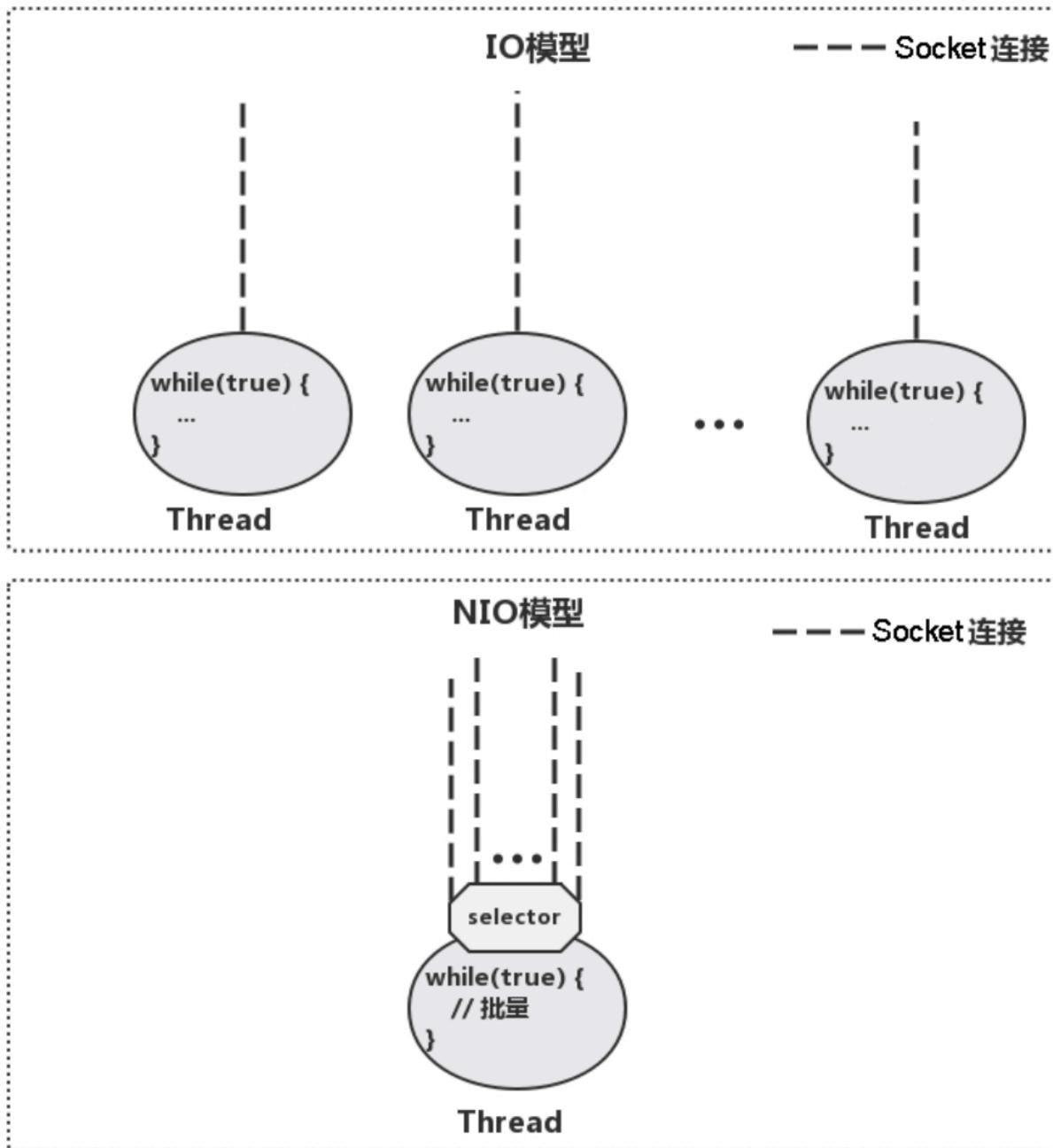
为了解决这3个问题，JDK在1.4版本之后提出了NIO。

2.2 NIO编程

网上有很多关于NIO的文章，这里不再深入分析。下面简单描述一下NIO是如何解决以上3个问题的。

2.2.1 线程资源受限

在NIO编程模型中，新来一个连接不再创建一个新线程，而是可以把这个连接直接绑定到某个固定的线程，然后这个连接所有的读写都由这个线程来负责，那么它是怎么做到的？我们用下图来对比一下IO与NIO。



如上图所示，在IO模型中，一个连接来了，会创建一个线程，对应一个while死循环，死循环的目的就是不断监测这个连接上是否有数据可以读。在大多数情况下，1万个连接里面同一时刻只有少量的连接有数据可读，因此，很多while死循环都白白浪费掉了，因为读不出数据。

而在NIO模型中，这么多while死循环转换为一个死循环，这个死循环由一个线程控制，那么NIO又是如何做到一个线程一个while死循环就能监测1万个连接是否有数据可读的呢？

这就是NIO模型中Selector的作用，一个连接来了之后，不会创建一个while死循环去监听是否有数据可读，而是直接把这条连接注册到Selector上。然后，通过检查这个Selector，就可以批量监测出有数据可读的连接，进而读取数据。下面我们举一个生活中非常简单的例子来说明IO与NIO的区别。

在一家幼儿园里，小朋友有上厕所的需求，小朋友都太小以至于你要问他要不要上厕所，他才会告诉你。幼儿园一共有100个小朋友，有两种方案可以解决小朋友上厕所的问题。

1.每个小朋友都配一个老师。每个老师都隔段时间询问小朋友是否要上厕所。如果要上，就领他去厕所，100个小朋友就需要100个老师来询问，并且每个小朋友上厕所的时候都需要一个老师领着他去，这就是IO模型，一个连接对应一个线程。

2.所有的小朋友都配同一个老师。这个老师隔段时间询问所有的小朋友是否有人要上厕所，然后每一时刻把所有要上厕所的小朋友批量领到厕所，这就是NIO模型。所有小朋友都注册到同一个老师，对应的就是所有的连接都注册到同一个线程，然后批量轮询。

这就是NIO模型解决线程资源受限问题的方案。在实际开发过程中，我们会开多个线程，每个线程都管理着一批连接，相对于IO模型中一个线程管理一个连接，消耗的线程资源大幅减少。

2.2.2 线程切换效率低下

由于NIO模型中线程数量大大降低，因此线程切换效率也大幅度提高。

2.2.3 IO读写面向流

IO读写是面向流的，一次性只能从流中读取一字节或者多字节，并且读完之后流无法再读取，需要自己缓存数据。而NIO的读写是面向Buffer的，可以随意读取里面任何字节数据，不需要自己缓存数据，只需要移动读写指针即可。

简单讲完了JDK NIO的解决方案之后，接下来我们使用NIO方案替换掉IO方案。先来看看，如果用JDK原生的NIO来实现服务端，该怎么做。

前方高能预警：以下代码可能会让你感觉极度不适，如有不适，请跳过。

NIOServer.java

```
/**
```

* @author 闪电侠

* /

```
public class NIOServer {  
  
    public static void main(String [] args) throws IOException {  
  
        Selector serverSelector = Selector.open();  
  
        Selector clientSelector = Selector.open();  
  
        new Thread(() -> {  
  
            try {  
  
                // 对应IO编程中的服务端启动  
  
                ServerSocketChannel listenerChannel = ServerSocketChannel.open();  
  
                listenerChannel.socket().bind(new InetSocketAddress(8000));  
  
                listenerChannel.configureBlocking(false);  
  
                listenerChannel.register(serverSelector, SelectionKey.OP_ACCEPT);  
  
                while (true) {  
  
                    // 监测是否有新连接，这里的1指阻塞的时间为 1ms  
  
                    if (serverSelector.select(1) > 0) {  
  
                        Set<SelectionKey> set = serverSelector.selectedKeys();  
  
                        Iterator<SelectionKey> keyIterator = set.iterator();  
  
                        while (keyIterator.hasNext()) {  
  
                            SelectionKey key = keyIterator.next();  
  
                            if (key.isAcceptable()) {  
  
                                // 处理新连接  
                            }  
                        }  
                    }  
                }  
            } catch (IOException e) {  
                e.printStackTrace();  
            }  
        }).start();  
    }  
}
```



```
// (2)批量轮询哪些连接有数据可读，这里的1指阻塞的时间为 1ms

if (clientSelector.select(1) > 0) {

Set<SelectionKey> set = clientSelector.selectedKeys();

Iterator<SelectionKey> keyIterator = set.iterator();

while (keyIterator.hasNext()) {

SelectionKey key = keyIterator.next();

if (key.isReadable()) {

try {

SocketChannel clientChannel = (SocketChannel) key.channel();

ByteBuffer byteBuffer = ByteBuffer.allocate(1024);

// (3)面向Buffer

clientChannel.read(byteBuffer);

byteBuffer.flip();

System.out.println(Charset.defaultCharset().newDecoder().

decode(byteBuffer)

.toString());

} finally {

keyIterator.remove();

```

相信大部分没有接触过NIO的读者应该会直接跳过代码来到这一行：原来使用JDK原生NIO的API实现一个简单的服务端通信程序如此复杂！

我们还是先对照NIO来解释一下核心思路。

1.NIO模型中通常会有两个线程，每个线程都绑定一个轮询器Selector。在这个例子中，serverSelector负责轮询是否有新连接，clientSelector负责轮询连接是否有数据可读。

2.服务端监测到新连接之后，不再创建一个新线程，而是直接将新连接绑定到clientSelector上，这样就不用IO模型中的1万个while循环死等，参见（1）。

3.clientSelector被一个while死循环包裹着，如果在某一时刻有多个连接有数据可读，那么通过clientSelector.select(1)方法可以轮询出来，进而批量处理，参见（2）。

4. 数据的读写面向Buffer，参见 (3)。

其他细节部分，因为实在是太复杂，所以笔者不再多讲，读者也不用对代码的细节深究到底。总之，强烈不建议直接基于JDK原生NIO来进行网络开发，下面是笔者总结

的原因。

1.JDK的NIO编程需要了解很多概念，编程复杂，对NIO入门非常不友好，编程模型不友好，`ByteBuffer`的API简直“反人类”。

2.对NIO编程来说，一个比较合适的线程模型能充分发挥它的优势，而JDK没有实现，需要自己实现，就连简单的自定义协议拆包都要自己实现。

3.JDK的NIO底层由Epoll实现，该实现饱受诟病的空轮询Bug会导致CPU占用率飙升至100%。

4.项目庞大之后，自行实现的NIO很容易出现各类Bug，维护成本较高，上面这些代码笔者都不能保证没有Bug。

正因为如此，客户端代码这里就省略了，读者可以直接使用`IOLClient.java`与`NIOServer.java`通信。

JDK的NIO犹如带刺的玫瑰，虽然美好，让人向往，但是使用不当会让你抓耳挠腮，痛不欲生，正因为如此，Netty横空出世！

2.3 Netty编程

Netty到底是何方神圣？

用一句简单的话来说就是：Netty封装了JDK的NIO，让你用得更方便，不用再写一大堆复杂的代码了。

用官方正式的话来说就是：Netty是一个异步事件驱动的网络应用框架，用于快速开发可维护的高性能服务端和客户端。

下面是笔者总结的使用Netty而不使用JDK原生NIO的原因。

1.使用JDK原生NIO需要了解太多概念，编程复杂，一不小心就Bug横飞。

2.Netty底层IO模型随意切换，而这一切只需要做微小的改动，改改参数，Netty可以直接从NIO模型变身为IO模型。

3.Netty自带的拆包/粘包、异常检测等机制让你从NIO的繁重细节中脱离出来，只需要关心业务逻辑即可。

4.Netty解决了JDK很多包括空轮询在内的Bug。

5.Netty底层对线程、Selector做了很多细小的优化，精心设计的Reactor线程模型可以做到非常高效的并发处理。

6.自带各种协议栈，让你处理任何一种通用协议都几乎不用亲自动手。

7.Netty社区活跃，遇到问题随时邮件列表或者Issue。

8.Netty已经历各大RPC框架、消息中间件、分布式通信中间件线上的广泛验证，健壮性无比强大。

这些原因看不懂没关系，在后续的章节中我们都可以学到。接下来我们用Netty来重新实现一下本章开篇的功能吧！

首先引入Maven依赖，本书后续Netty都基于4.1.6.Final版本。

```
<dependency>
    <groupId>io.netty</groupId>
    <artifactId>netty-all</artifactId>
    <version>4.1.6.Final</version>
</dependency>
```

然后是服务端实现部分。

NettyServer.java

```
 /**
 * @author 闪电侠
 */
public class NettyServer {
    public static void main(String [] args) {
        ServerBootstrap serverBootstrap = new ServerBootstrap();
        NioEventLoopGroup boss = new NioEventLoopGroup();
```

```
NioEventLoopGroup worker = new NioEventLoopGroup();

serverBootstrap

.group(boss, worker)

.channel(NioServerSocketChannel.class)

.childHandler(new ChannelInitializer<NioSocketChannel>()  {

protected void initChannel(NioSocketChannel ch)  {

ch.pipeline().addLast(new StringDecoder());

ch.pipeline().addLast(new SimpleChannelInboundHandler<String>()  {

@Override

protected void channelRead0(ChannelHandlerContext ctx, String msg)  {

System.out.println(msg);

}

}

}

}

)

.bind(8000);

}

}

}
```

这么一小段代码就实现了我们前面NIO编程中的所有功能，包括服务端启动、接收新连接、打印客户端传来的数据，怎么样？是不是比JDK原生NIO编程简洁许多？

初学Netty的时候，由于大部分人对NIO编程缺乏经验，因此，将Netty里的概念与IO模型结合起来可能更好理解。

1.boss对应IOServer.java中的负责接收新连接的线程，主要负责创建新连接。

2.worker对应IOServer.java中的负责读取数据的线程，主要用于读取数据及业务逻辑处理。

剩下的逻辑笔者在后面的内容中会详细分析，读者可以先把这段代码复制到自己的IDE里，然后运行main函数。

下面是客户端NIO的实现部分。

NettyClient.java

```
 /**
 * @author 闪电侠
 */
public class NettyClient {
    public static void main(String [] args) throws InterruptedException
    {
        Bootstrap bootstrap = new Bootstrap();
        NioEventLoopGroup group = new NioEventLoopGroup();
        bootstrap.group(group)
            .channel(NioSocketChannel.class)
            .handler(new ChannelInitializer<Channel>() {
                @Override
                protected void initChannel(Channel ch) {
                    ch.pipeline().addLast(new StringEncoder());
                }
            });
    }
}
```

```
Channel channel = bootstrap.connect("127.0.0.1", 8000).channel();

while (true) {

    channel.writeAndFlush(new Date() + ": hello world! ");

    Thread.sleep(2000);

}

}

}
```

在客户端程序中，group对应了IOClient.java中main函数起的线程，剩下的逻辑在后面的内容中会详细分析，现在你要做的事情就是把这段代码复制到你的IDE里，然后运行main函数，最后回到NettyServer.java的控制台，你会看到效果。

使用Netty之后是不是觉得整个世界都变美好了？一方面，Netty对NIO封装得如此完美，写出来的代码非常优雅；另一方面，使用Netty之后，网络通信的性能问题几乎不用操心，尽情地让Netty“榨干”你的CPU吧。

第3章

Netty开发环境配置

本章介绍Netty开发环境的搭建，笔者假设读者已经有了Java编程需要的环境。如果读者已经安装过Maven、Git、IntelliJ IDEA环境，建议直接看本章末尾的“如何使用本书的代码”。

3.1 Maven

Maven是一个基于对象模型来管理项目构建的项目管理工具，通过配置文件pom.xml来配置jar包，相对于传统复制jar包的方式，管理依赖更为方便，如果你没有安装过Maven，下面的指导将带你一起安装。

3.1.1 下载

首先，到Apache官网下载Maven，由于Maven也是使用Java编写的，所以不同操作系统下载的Maven zip包是一样的，这里选择最新的版本：apache-maven-版本号-src.zip，下载到本地之后解压，接下来看不同的操作系统配置。

3.1.2 配置和验证

Windows

- 1.假定我们将文件夹解压到D:\maven，该目录下有bin、lib等目录。
- 2.通过“我的电脑”->“属性”->“高级系统设置”->“环境变量”->“系统变量”->“新建”新建一个环境变量，变量名为M2_HOME，值为D:\maven。
- 3.找到变量名字为Path的环境变量，在变量值的尾部加入“;%M2_HOME%\bin;”，这里需要注意前面的分号。

最后，打开Dos窗口，输入mvn -v，如果出来版本号相关的信息，则说明我们的Maven已经安装成功了。

Linux && Mac

假定我们将文件夹解压到/usr/local/maven，该目录下有bin、lib等目录；接下来，和Windows系统一样，需要配置环境变量。我们打开etcprofile文件，在尾部加入下面两

行代码。

```
export MAVEN_HOME=/usr/local/maven  
  
PATH=$JAVA_HOME/bin: $MAVEN_HOME/bin: $PATH
```

然后，在命令行执行source *etcprofile*，让配置生效；接下来，通过mvn -v命令来验证是否生效，如果出来的是版本号相关的信息，则说明Maven已经安装成功了。

3.2 Git

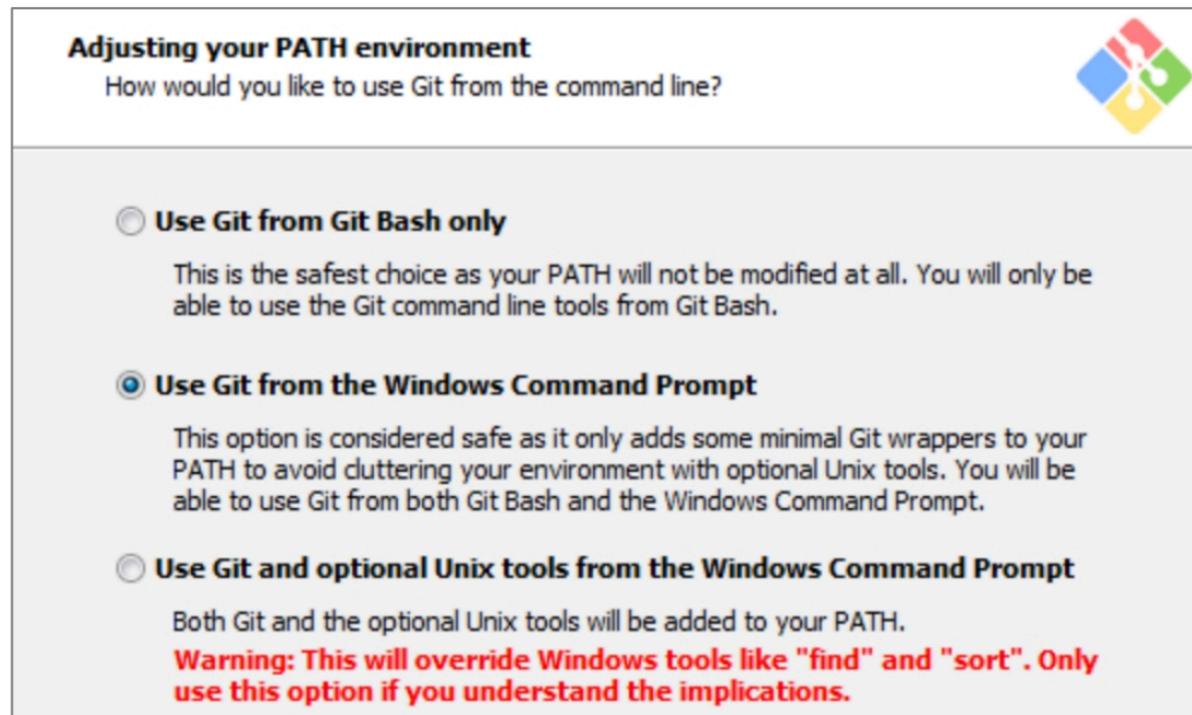
Git是一个版本管理工具，本书代码使用Git来做版本控制，每个章节的代码都是一个Git分支，方便读者循序渐进地学习。我们来看一下如何安装配置Git。

3.2.1 下载与安装

Windows

1.在gitforwindows.org下载最新版本的Windows Git。

2.下载完成之后，双击exe文件，只需要一直单击“下一步”按钮，安装即可。其中有一步需要注意一下，如下图所示。



该步骤为调整环境变量，我们选择中间的一项，继续单击“下一步”按钮，直到安装成功。

3.安装成功之后，在任意目录上单击鼠标右键，选择“Git Bash Here”这一项，输入git，如果出来提示，则表明安装成功。

Linux & Mac

1.如果你使用的是Debian或Ubuntu，那么直接使用一条命令sudo apt-get install git即可完成安装；如果是centOS版本，则在命令行执行yum install -y git即可完成安装。

2.Mac系统自带Git，不过默认没有安装，你需要运行xcode，然后选择菜单“xcode”->“Preferences”，选择“Downloads”这个Tab页面，再选择“Command Line Tools”，单击“Install”按钮即可完成安装。

3.2.2 配置

最后，我们通过在命令行依次输入以下命令来配置你的名字和邮箱，这样在提交代码的时候就能知道作者的信息。

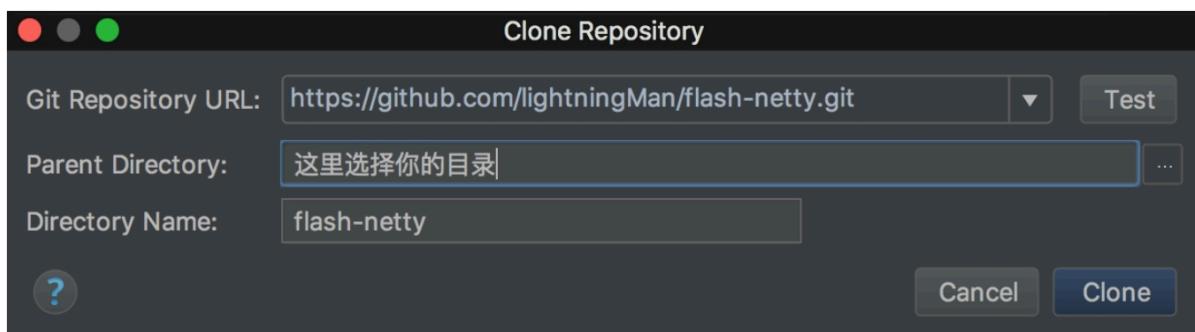
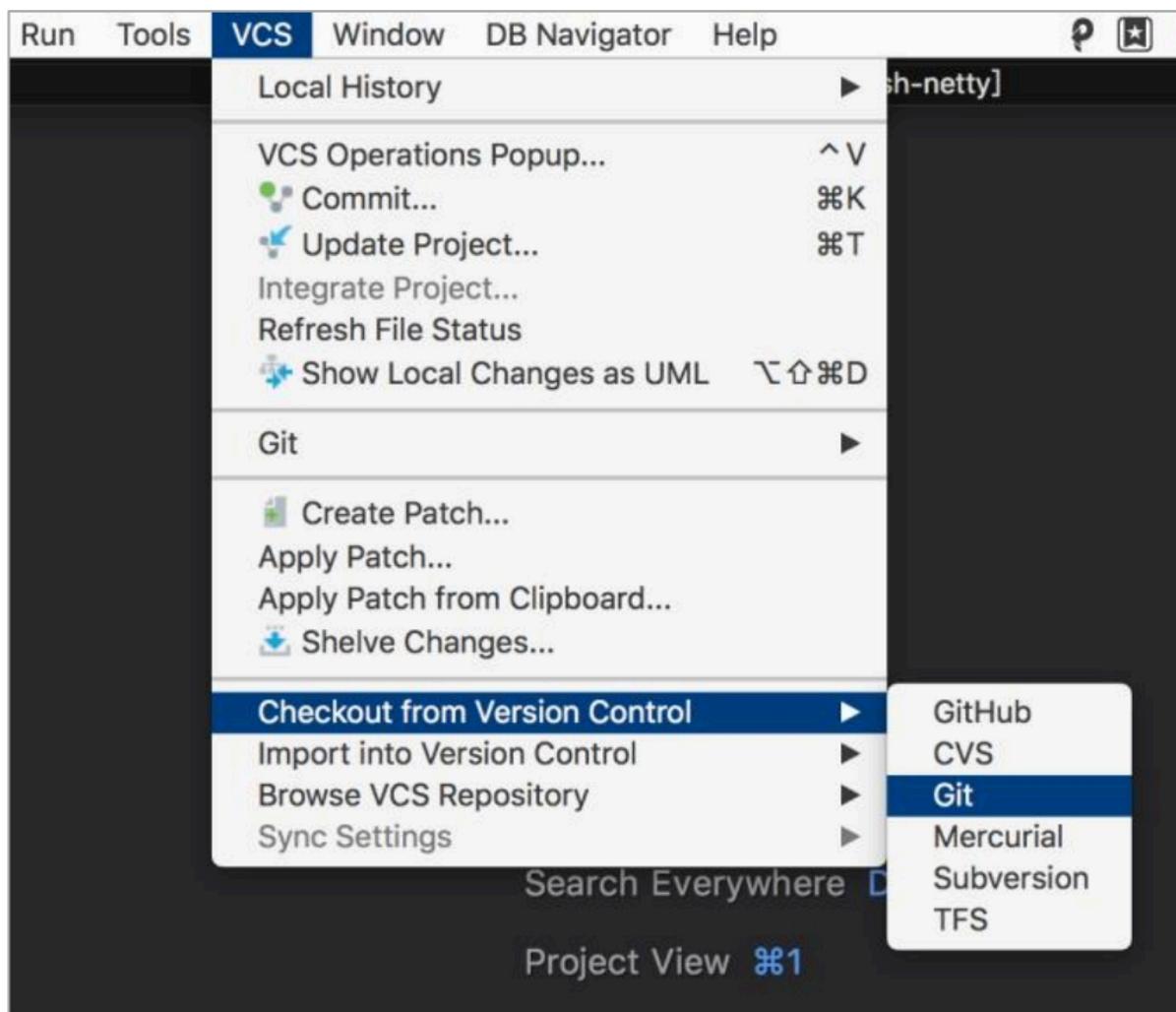
```
git config --global user.name "Your Name"  
git config --global user.email "email@example.com"
```

3.3 IntelliJ IDEA

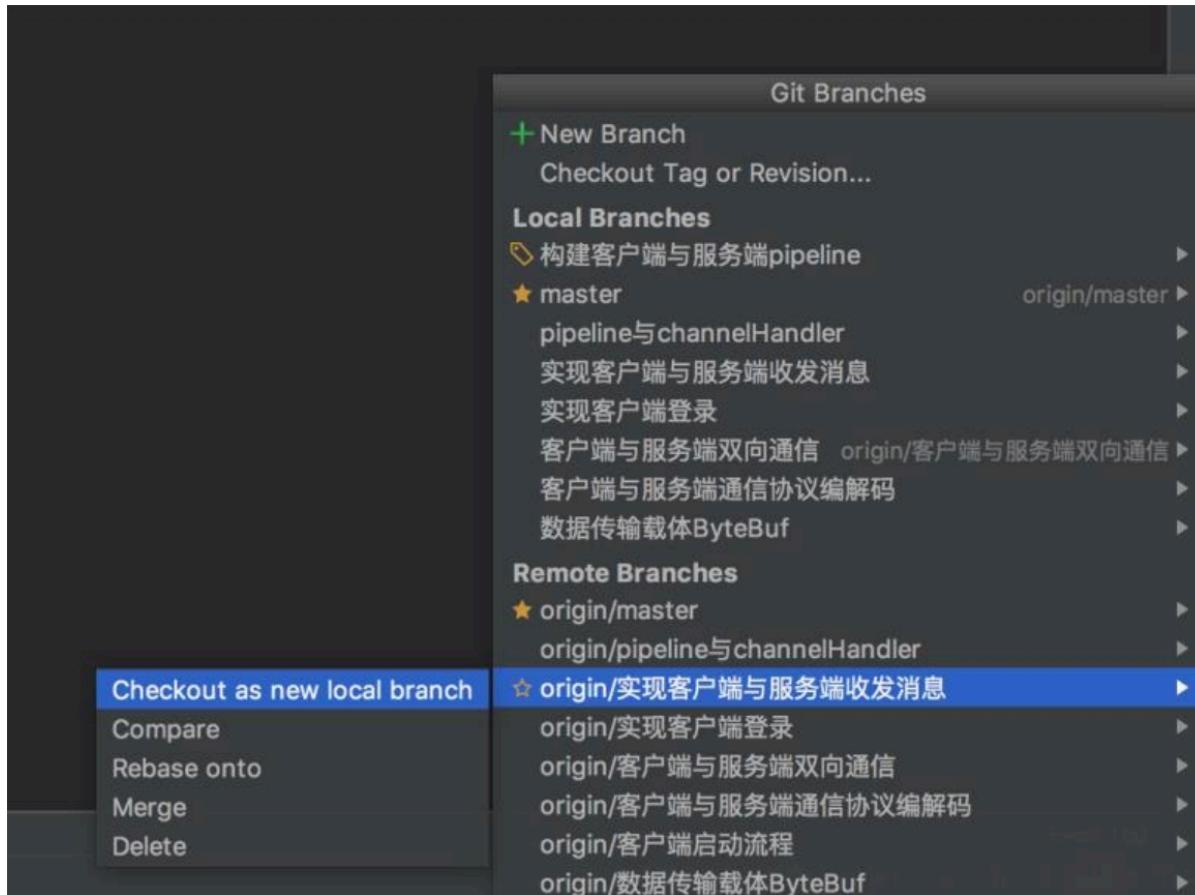
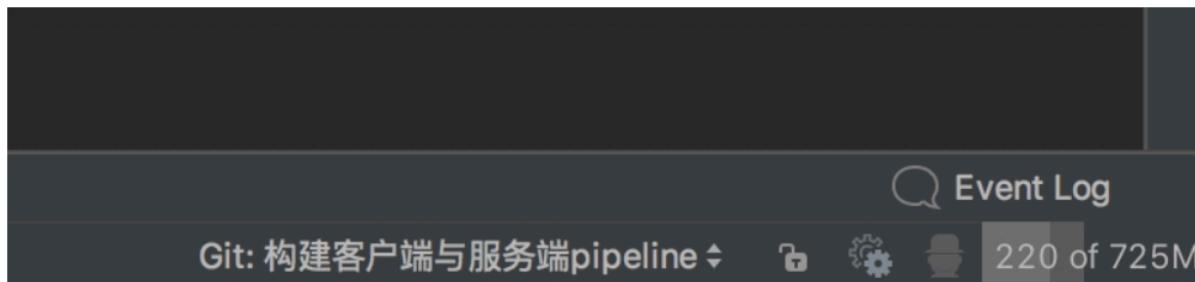
本书使用IntelliJ IDEA作为集成开发环境。当然，如果你非常熟悉Eclipse，也可以使用Eclipse。对于想入门学习IntelliJ IDEA的读者，笔者之前录制的一个免费视频可以奉献给大家，请通过“读者服务”扫码获取，详细的安装过程和介绍，该视频里均有。

接下来我们看一下如何使用本书的代码。

首先，我们通过下图所示的步骤将代码仓库导入本地。



代码复制到本地之后，在IntelliJ IDEA右下角切换相应的分支，即可找到每一节对应的完整代码，如下图所示。



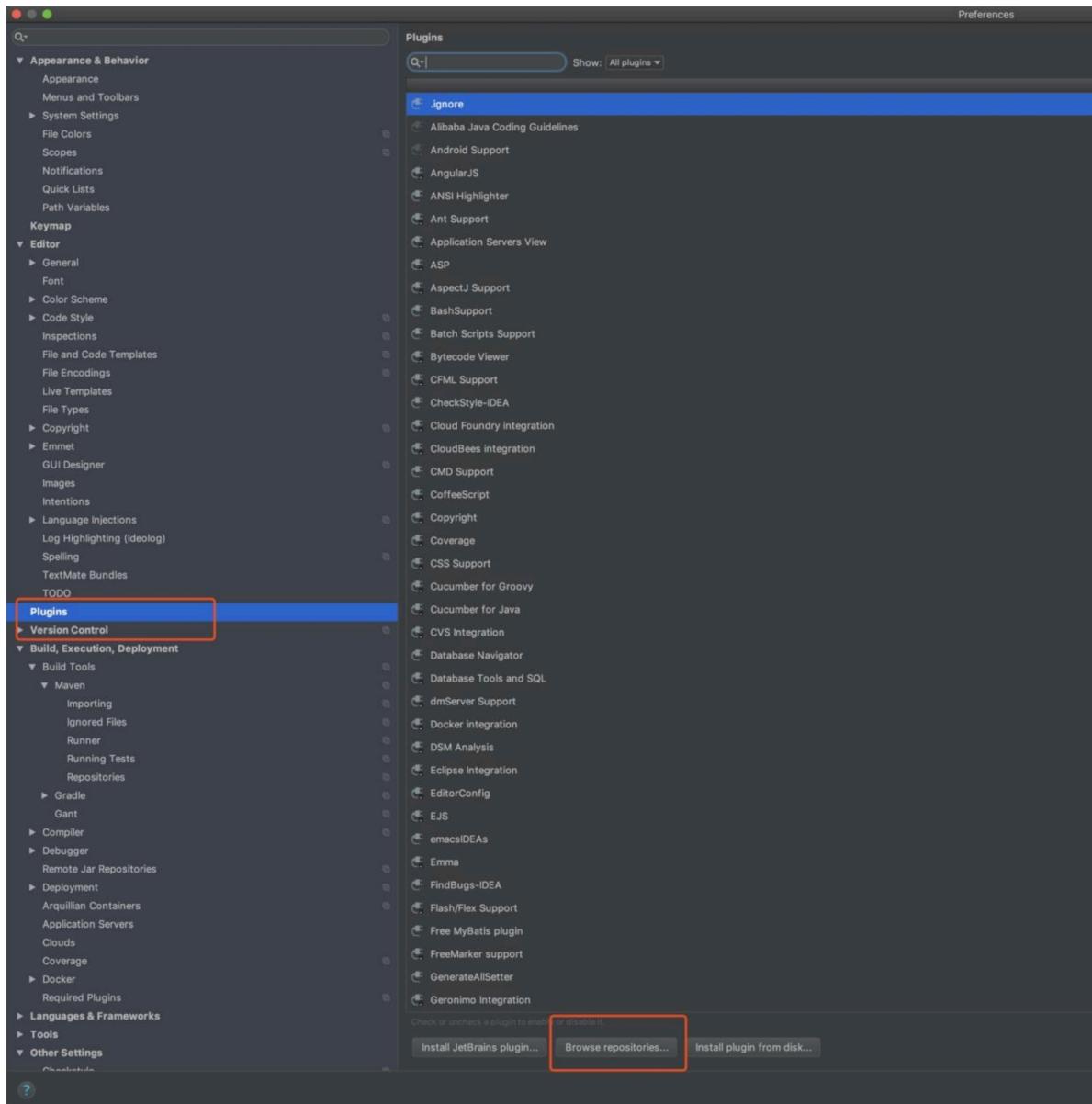
```
private static final int PORT = 8000;
public static void main(String[] args) {
    NioEventLoopGroup bossGroup = new NioEventLoopGroup();
    NioEventLoopGroup workerGroup = new NioEventLoopGroup();
    final ServerBootstrap serverBootstrap = new ServerBootstrap();
    serverBootstrap
        .group(bossGroup, workerGroup)
        .channel(NioServerSocketChannel.class)
        .option(ChannelOption.SO_BACKLOG, 1024)
        .childOption(ChannelOption.SO_KEEPALIVE, true)
        .childHandler((ChannelInitializer<Channel> ) ch -> {
            ch.pipeline().addLast(new PacketDecoder());
            ch.pipeline().addLast(new LoginRequestHandler());
            ch.pipeline().addLast(new MessageRequestsHandler());
            ch.pipeline().addLast(new PacketEncoder());
        });
    serverBootstrap.bind(PORT);
}
private static void bind(final ServerBootstrap serverBootstrap, final int port) {
    serverBootstrap.bind(port).addListener(future -> {
        if (future.isSuccess()) {
            System.out.println(new Date() + ": 端口[" + port + "]绑定成功!");
        } else {
            System.err.println("端口[" + port + "]绑定失败!");
        }
    });
}
```

由于在代码里，笔者使用了lombok自动生成getter、setter及构造函数，需要在IntelliJ IDEA中安装插件，否则代码会报红，具体安装可以参考下图所示的步骤。

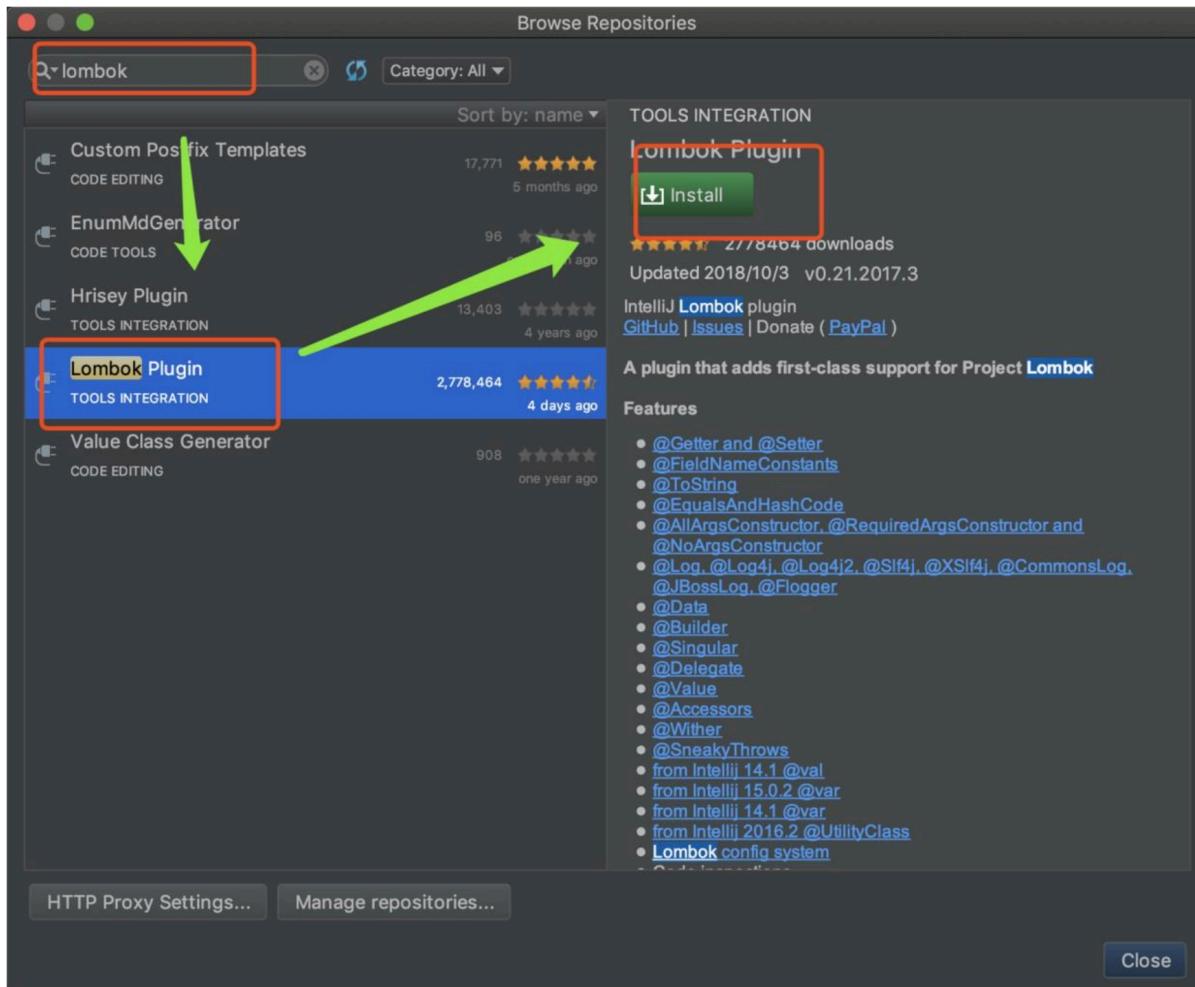
首先调出配置。



然后找到IDEA插件相关的配置。



接着在弹出来的窗口中输入lombok。



最后单击“Install”按钮安装，之后重启IntelliJ IDEA即可。

第4章

服务端启动流程

这一章，我们学习如何使用Netty来启动一个服务端应用程序。

4.1 服务端启动最小化代码

以下是服务端启动的一个非常精简的Demo。

NettyServer.java

```
public class NettyServer {  
  
    public static void main(String [] args) {  
  
        NioEventLoopGroup bossGroup = new NioEventLoopGroup();  
  
        NioEventLoopGroup workerGroup = new NioEventLoopGroup();  
  
        ServerBootstrap serverBootstrap = new ServerBootstrap();  
  
        serverBootstrap  
  
            .group(bossGroup, workerGroup)  
            .channel(NioServerSocketChannel.class) .childHandler(new  
ChannelInitializer<NioSocketChannel>() {  
  
    protected void initChannel(NioSocketChannel ch) {  
  
    }  
});  
  
serverBootstrap.bind(8000);  
  
    }  
}
```

- 上述代码首先创建了两个NioEventLoopGroup，这两个对象可以看作传统IO编程模型的两大线程组，bossGroup表示监听端口，接收新连接的线程组；workerGroup表示处理每一个连接的数据读写的线程组。用生活中的例子来讲就是，一个工厂要运作，必然要有一个老板负责从外面接活，然后有很多员工，负责具体干活。老板就是bossGroup，员工们就是workerGroup，bossGroup接收完连接，交给workerGroup去处理。
- 其次创建了一个引导类ServerBootstrap，这个类将引导服务端的启动工作。
- 通过.group(bossGroup,workerGroup)给引导类配置两大线程组，这个引导类的线程模型也就定型了。
- 然后指定服务端的IO模型为NIO，上述代码通过.channel(NioServerSocketChannel.class)来指定IO模型，也可以有其他选择。如果你想指定IO模型为BIO，那么这里配置上OioServerSocketChannel.class类型即可。当然通常我们也不会这么做，因为Netty的优势就在于NIO。
- 接着调用childHandler()方法，给这个引导类创建一个ChannelInitializer，主要是定义后续每个连接的数据读写，对于业务处理逻辑，不理解也没关系，后面我们会详细分析。在ChannelInitializer这个类中，有一个泛型参数NioSocketChannel，这个类就是Netty对NIO类型连接的抽象，而前面的NioServerSocketChannel也是对NIO类型连接的抽象，NioServerSocketChannel和NioSocketChannel的概念可以与BIO编程模型中的ServerSocket和Socket两个概念对应。

最小化参数配置到这里就完成了，总结一下就是，要启动一个Netty服务端，必须要指定三类属性，分别是线程模型、IO模型、连接读写处理逻辑。有了这三者，之后再调用bind(8000)，就可以在本地绑定一个8000端口启动服务端。以上这段代码，读者可以直接复制到自己的IDE中运行。

4.2 自动绑定递增端口

上面代码绑定了8000端口，接下来我们实现一个稍微复杂点的逻辑。我们指定一个起始端口号，比如1000；然后从1000端口往上找一个端口，直到这个端口能够绑定成功。比如1000端口不可用，我们就尝试绑定1001端口，然后1002端口，以此类推。

serverBootstrap.bind(8000)方法是一个异步方法，调用之后是立即返回的，它的返回值是一个ChannelFuture。我们可以给这个ChannelFuture添加一个监听器GenericFutureListener，然后在GenericFutureListener的operationComplete方法里，监听端口是否绑定成功。下面是监听端口是否绑定成功的代码片段。

```
serverBootstrap.bind(8000).addListener(new
GenericFutureListener<Future<? super
Void>>() {
    public void operationComplete(Future<? super Void> future) {
        if (future.isSuccess()) {
            System.out.println("端口绑定成功! ");
        } else {
            System.err.println("端口绑定失败! ");
        }
    }
});
```

接下来就可以从1000端口开始，往上找端口号，直到端口绑定成功。我们要做的就是在if (future.isSuccess())的else逻辑里重新绑定一个递增的端口。我们从这段绑定逻辑中抽取出一个bind方法。

```
private static void bind(final ServerBootstrap serverBootstrap, final
int port) {
    serverBootstrap.bind(port).addListener(new
GenericFutureListener<Future<? super
Void>>() {
    public void operationComplete(Future<? super Void> future) {
        if (future.isSuccess()) {
            System.out.println("端口 [" + port + "] 绑定成功! ");
        } else {
```

```
System.err.println("端口 [" + port + "] 绑定失败! ");

bind(serverBootstrap, port + 1);

}

}

}

});
```

上述代码中最关键的就是在端口绑定失败之后，重新调用自身方法，并且把端口号加一，这样，在我们的主流程里面就可以直接调用。

```
bind(serverBootstrap, 1000)
```

读者可以自行修改代码，运行之后看到效果，最终会发现，端口成功绑定在了1024。从1000开始到1023，端口均绑定失败，这是因为在笔者的Mac系统下，1023以下的端口号都被系统保留了，需要ROOT权限才能绑定。

以上就是自动绑定递增端口的逻辑。接下来，我们一起学习一下，服务端启动引导类ServerBootstrap除了指定线程模型、IO模型、连接读写处理逻辑，还可以做哪些事情？

4.3 服务端启动的其他方法

4.3.1 handler()方法

```
serverBootstrap.handler(new ChannelInitializer<NioServerSocketChannel>
() {
    protected void initChannel(NioServerSocketChannel ch) {
        System.out.println("服务端启动中");
    }
})
```

handler()方法可以和前面分析的childHandler()方法对应起来：childHandler()方法用于指定处理新连接数据的读写处理逻辑；handler()方法用于指定在服务端启动过程中的一些逻辑，通常情况下用不到这个方法。

4.3.2 attr()方法

```
serverBootstrap.attr(AttributeKey.newInstance("serverName"), "nettyServer")
```

attr()方法可以给服务端Channel，也就是NioServerSocketChannel指定一些自定义属性，然后通过channel.attr()取出这个属性。比如，上面的代码可以指定服务端Channel的serverName属性，属性值为nettyServer，其实质是给NioServerSocketChannel维护一个Map而已，通常情况下也用不上这个方法。

4.3.3 childAttr()方法

除了可以给服务端Channel即NioServerSocketChannel指定一些自定义属性，我们还可以给每一个连接都指定自定义属性。

```
serverBootstrap.childAttr(AttributeKey.newInstance("clientKey"), "clientValue")
```

上面的childAttr()方法可以给每一个连接都指定自定义属性，后续我们可以通过channel.attr()方法取出该属性。

4.3.4 option()方法

option()方法可以给服务端Channel设置一些TCP参数，最常见的就是so_backlog，设置如下。

```
serverBootstrap.option(ChannelOption.SO_BACKLOG, 1024)
```

这个设置表示系统用于临时存放已完成三次握手的请求的队列的最大长度，如果连接建立频繁，服务器处理创建新连接较慢，则可以适当调大这个参数。

4.3.5 childOption()方法

childOption()方法可以给每个连接都设置一些TCP参数。

```
serverBootstrap
```

```
.childOption(ChannelOption.SO_KEEPALIVE, true)  
.childOption(ChannelOption.TCP_NODELAY, true)
```

上述代码中设置了两种TCP参数，其中：

- ChannelOption.SO_KEEPALIVE表示是否开启TCP底层心跳机制，true表示开启。
- ChannelOption.TCP_NODELAY表示是否开启Nagle算法，true表示关闭，false表示开启。通俗地说，如果要求高实时性，有数据发送时就马上发送，就设置为关闭；

如果需要减少发送次数，减少网络交互，就设置为开启。

其他参数这里就不一一讲解了，读者有兴趣可以自行研究。

4.4 总结

- 在本章中，我们首先学习了Netty服务端启动的流程，一句话总结就是：首先创建一个引导类，然后给它指定线程模型、IO模型、连接读写处理逻辑，绑定端口之后，服务端就启动起来了。
- 然后我们学习到bind方法是异步的，可以通过这个异步机制来实现递增端口绑定。
- 最后我们讨论了Netty服务端启动的其他方法，主要包括给服务端Channel或者客户端Channel设置属性值、设置底层TCP参数。

如果你觉得这个过程比较简单，想深入了解服务端启动的底层原理，可参考本书第21章。

第5章

客户端启动流程

上一章，我们已经学习了Netty服务端启动流程；这一章，我们来学习Netty客户端启动流程。

5.1 客户端启动Demo

对于客户端的启动来说，和服务端的启动类似，依然需要线程模型、IO模型，以及IO业务处理逻辑三大参数。下面我们来看一下客户端启动的标准流程。

NettyClient.java

```
public class NettyClient {  
  
    public static void main(String [] args) {  
  
        NioEventLoopGroup workerGroup = new NioEventLoopGroup();  
  
        Bootstrap bootstrap = new Bootstrap();  
  
        bootstrap  
  
        // 1.指定线程模型  
  
        .group(workerGroup) // 2.指定 IO 类型为 NIO  
  
        .channel(NioSocketChannel.class) // 3.IO 处理逻辑  
  
        .handler(new ChannelInitializer<SocketChannel>() {  
  
            @Override public void initChannel(SocketChannel ch) {  
  
            }  
  
        } );  
  
        // 4.建立连接
```

```
bootstrap.connect("juejin.cn", 80).addListener(future -> {  
  
    if (future.isSuccess()) {  
  
        System.out.println("连接成功! ");  
  
    } else {  
  
        System.err.println("连接失败! ");  
  
    }  
});  
  
}  
}
```

从上面的代码可以看到，客户端启动的引导类是Bootstrap，负责启动客户端和连接服务端；而在服务端启动的时候，这个引导类是ServerBootstrap。引导类创建完成之后，客户端启动的流程如下。

1.与服务端的启动一样，需要给它指定线程模型，驱动连接的数据读写，这个线程的概念可以和第1章中IOClient.java创建的线程联系起来。

2. 指定IO模型为NioSocketChannel，表示IO模型为NIO。当然，你可以设置IO模型为OioSocketChannel，但是通常不会这么做，因为Netty的优势在于NIO。

3.给引导类指定一个Handler，主要定义连接的业务处理逻辑，不理解没关系，在后面会详细分析。

4.配置完线程模型、IO模型、业务处理逻辑之后，调用connect方法进行连接，可以看到connect方法有两个参数，第一个参数可以填写IP或者域名，第二个参数填写端口号。由于connect方法返回的是一个Future，也就是说这个方法是异步的，通过addListener方法可以监听连接是否成功，进而打印连接信息。

到了这里，一个客户端启动的Demo就完成了，其实只要和客户端Socket编程模型对应起来，这里的三个概念就会显得非常简单。读者如果忘掉了，可以先回顾一下第1章的IOClient.java，再来看这里的启动流程。

5.2 失败重连

在网络情况差的情况下，客户端第一次连接可能会连接失败，这个时候我们可能会尝试重连。重连的逻辑写在连接失败的逻辑块里。

```
bootstrap.connect("meituan.com", 80).addListener(future -> {

    if (future.isSuccess()) {
        System.out.println("连接成功! ");
    } else {
        System.err.println("连接失败! ");
        // 重连
    }
});
```

在重连的时候，依然调用同样的逻辑。因此，我们把建立连接的逻辑先抽取出来，然后在重连的时候，递归调用自身。

```
private static void connect(Bootstrap bootstrap, String host, int port) {
    bootstrap.connect(host, port).addListener(future -> {
        if (future.isSuccess()) {
            System.out.println("连接成功! ");
        } else {
            System.err.println("连接失败，开始重连");
            connect(bootstrap, host, port);
        }
    });
}
```

```
    } );  
  
    }  
}
```

上面这一段便是带有自动重连功能的逻辑，可以看到在连接失败的时候，会调用自身进行重连。

但是，在通常情况下，连接失败不会立即重连，而是通过一个指数退避的方式，比如每隔1秒、2秒、4秒、8秒，以2的幂次来建立连接，到达一定次数之后就放弃连接。接下来我们实现这段逻辑，默认重试5次。

```
connect(bootstrap, "meituan.com", 80, MAX_RETRY);  
  
private static void connect(Bootstrap bootstrap, String host, int  
port, int retry) {  
  
    bootstrap.connect(host, port).addListener(future -> {  
  
        if (future.isSuccess()) {  
  
            System.out.println("连接成功！");  
  
        } else if (retry == 0) {  
  
            System.err.println("重试次数已用完，放弃连接！");  
  
        } else {  
  
            // 第几次重连  
  
            int order = (MAX_RETRY - retry) + 1;  
  
            // 本次重连的间隔  
  
            int delay = 1 << order;  
  
            System.err.println(new Date() + ": 连接失败，第" + order + "次重  
连.....");  
        }  
    });  
}  
}
```

```
        bootstrap.config().group().schedule(() -> connect(bootstrap, host,
port, retry--), delay, TimeUnit.SECONDS);

    }

}

} );
```

从上面的代码可以看到，通过判断连接是否成功及剩余的重试次数，分别执行不同的逻辑。

- 1.如果连接成功，则打印连接成功的消息。
- 2.如果连接失败但重试次数已经用完，则放弃连接。
- 3.如果连接失败但重试次数仍然没有用完，则计算下一次重连间隔delay，然后定期重连。

在上面的代码中，我们看到，定时任务调用的是
bootstrap.config().group().schedule()，其中bootstrap.config()这个方法返回的是
BootstrapConfig，它是对Bootstrap配置参数的抽象，然后bootstrap.config().group()
返回的就是我们在一开始配置的线程模型workerGroup，调用workerGroup的
schedule方法即可实现定时任务逻辑。

在schedule方法块里，前四个参数原封不动地传递，最后一个重试次数参数减掉1，
就是下一次建立连接时的上下文信息。读者可以自行修改代码，更改到一个连接不
上的服务端Host或者Port，查看控制台日志就可以看到5次重连日志。

以上就是实现指数退避的客户端重连逻辑。接下来，我们一起学习一下，客户端启
动过程中的引导类Bootstrap除了指定线程模型、IO模型、连接读写处理逻辑，还可
以做哪些事情？

5.3 客户端启动的其他方法

5.3.1 attr()方法

bootstrap.attr(AttributeKey.newInstance("clientName"), "nettyClient")
attr()方法可以为客户端Channel也就是NioSocketChannel绑定自定义属性，然后通过
channel.attr()方法取出这个属性。比如，上面的代码可以指定客户端Channel的

clientName属性，属性值为nettyClient，其实就是为NioSocketChannel维护一个Map而已。后续在NioSocketChannel通过参数传来传去的时候，就可以通过它来取出设置的属性，非常方便。

5.3.2 option()方法

Bootstrap

```
.option(ChannelOption.CONNECT_TIMEOUT_MILLIS, 5000)  
.option(ChannelOption.SO_KEEPALIVE, true)  
.option(ChannelOption.TCP_NODELAY, true) option()方法可以为连接设置一些  
TCP底层相关的属性，比如上面的代码中，我们设置了3种TCP属性，其中：
```

- ChannelOption.CONNECT_TIMEOUT_MILLIS表示连接的超时时间，超过这个时间，如果仍未连接到服务端，则表示连接失败。
- ChannelOption.SO_KEEPALIVE表示是否开启TCP底层心跳机制，true表示开启。
- ChannelOption.TCP_NODELAY表示是否开始Nagle算法，true表示关闭，false表示开启。通俗地说，如果要求高实时性，有数据发送时就马上发送，就设置为true；如果需要减少发送次数，减少网络交互，就设置为false。

其他参数这里就不一一讲解了，读者有兴趣可以去自行研究。

5.4 总结

- 本章中我们首先学习了Netty客户端启动的流程，一句话总结就是：首先创建一个引导类，然后为它指定线程模型、IO模型、连接读写处理逻辑，连接上特定主机和端口后，客户端就启动起来了。
- 然后我们学习到connect方法是异步的，可以通过异步回调机制来实现指数退避重连逻辑。
- 最后我们讨论了Netty客户端启动的其他方法，主要包括给客户端Channel绑定自定义属性值、设置底层TCP参数。

5.5 思考

与服务端启动相比，客户端启动的引导类少了哪些方法，为什么不需要这些方法？

第6章

客户端与服务端双向通信

在前面两章中，我们学习了服务端启动与客户端启动的流程。熟悉了这两个流程之后，就可以建立服务端与客户端之间的连接了。本章我们用一个Demo来了解服务端和客户端是如何通信的。

本章要实现的功能是：在客户端连接成功之后，向服务端写一段数据；服务端收到数据之后打印，并向客户端返回一段数据。这里展示的是核心代码，完整代码请参考代码仓库对应的章节。

6.1 客户端发送数据到服务端

在客户端启动流程这一章，读者已经了解到客户端相关的数据读写逻辑是通过Bootstrap的handler()方法指定的。

```
.handler(new ChannelInitializer<SocketChannel>() {
    @Override public void initChannel(SocketChannel ch) {
        // 指定连接数据读写逻辑
    }
});
```

接下来在initChannel()方法里给客户端添加一个逻辑处理器，其作用是负责向服务端写数据。

```
.handler(new ChannelInitializer<SocketChannel>() {
    @Override public void initChannel(SocketChannel ch) {
        ch.pipeline().addLast(new FirstClientHandler());
    }
});
```

1.ch.pipeline()返回的是和这条连接相关的逻辑处理链，采用了责任链模式。

2.调用addLast()方法添加一个逻辑处理器，逻辑处理器的作用就是，在客户端建立连接成功之后，向服务端写数据。下面是这个逻辑处理器相关的代码。

```
public class FirstClientHandler extends ChannelInboundHandlerAdapter {  
  
    @Override public void channelActive(ChannelHandlerContext ctx) {  
  
        System.out.println(new Date() + ": 客户端写出数据");  
  
        // 1. 获取数据  
  
        ByteBuf buffer = getByteBuf(ctx);  
  
        // 2. 写数据  
  
        ctx.channel().writeAndFlush(buffer);  
  
    }  
  
    private ByteBuf getByteBuf(ChannelHandlerContext ctx) {  
  
        // 1. 获取二进制抽象 ByteBuf ByteBuf buffer = ctx.alloc().buffer();  
  
        // 2. 准备数据，指定字符串的字符集为 UTF-8  
  
        byte [] bytes = "你好，闪电侠!".getBytes(Charset.forName("utf-8"));  
  
        // 3. 填充数据到 ByteBuf buffer.writeBytes(bytes);  
  
        return buffer;  
  
    }  
}
```

1.这个逻辑处理器继承自ChannelInboundHandlerAdapter，覆盖了channelActive()方法，这个方法会在客户端连接建立成功之后被调用。

2.客户端连接建立成功之后，调用channelActive()方法。在这个方法里，我们编写向服务端写数据的逻辑。

3.写数据的逻辑分为三步：首先需要获取一个Netty对二进制数据的抽象ByteBuf。在上面代码中，ctx.alloc()获取到一个ByteBuf的内存管理器，其作用就是分配一个ByteBuf。然后把字符串的二进制数据填充到ByteBuf，这样就获取到Netty需要的数据格式。最后调用ctx.channel().writeAndFlush()把数据写到服务端。

以上就是客户端启动之后，向服务端写数据的逻辑。可以看到，和传统的Socket编程不同的是，Netty里的数据是以ByteBuf为单位的，所有需要写出的数据都必须放到一个ByteBuf中。数据的写出如此，数据的读取亦如此。接下来我们看一下服务端是如何读取这段数据的。

6.2 服务端读取客户端数据

在服务端启动流程这一章，我们提到，服务端相关的数据处理逻辑是通过ServerBootstrap的childHandler()方法指定的。

```
.childHandler(new ChannelInitializer<NioSocketChannel>() {
    protected void initChannel(NioSocketChannel ch) {
        // 指定连接数据读写逻辑
    }
});
```

现在，我们在initChannel()方法里给服务端添加一个逻辑处理器，这个处理器的作用就是负责读取客户端发来的数据。

```
.childHandler(new ChannelInitializer<NioSocketChannel>() {
    protected void initChannel(NioSocketChannel ch) {
        ch.pipeline().addLast(new FirstServerHandler());
    }
});
```

这个方法里的逻辑和客户端侧类似，获取服务端侧关于这个连接的逻辑处理链Pipeline，然后添加一个逻辑处理器，负责读取客户端发来的数据。

```
public class FirstServerHandler extends ChannelInboundHandlerAdapter {  
  
    @Override public void channelRead(ChannelHandlerContext ctx, Object msg) {  
  
        ByteBuf byteBuf = (ByteBuf) msg;  
  
        System.out.println(new Date() + ": 服务端读到数据 -> " +  
byteBuf.toString(Charset.forName("utf-8")));  
  
    }  
  
}
```

服务端侧的逻辑处理器同样继承自ChannelInboundHandlerAdapter。与客户端不同的是，这里覆盖的方法是channelRead()，这个方法在接收到客户端发来的数据之后被回调。

这里的msg参数指的就是Netty里数据读写的载体，为什么这里不直接是ByteBuf，而需要强转一下呢？我们后面会分析到。这里我们强转之后，调用byteBuf.toString()就能够获得客户端发来的字符串数据。

我们先运行服务端，再运行客户端，下面两个图分别是服务端控制台和客户端控制台的输出。

服务端

```
端口[8000]绑定成功!  
Sat Aug 04 09:27:40 CST 2018: 服务端读到数据 -> 你好，闪电侠！
```

客户端

```
连接成功!  
Sat Aug 04 09:19:23 CST 2018: 客户端写出数据
```

到目前为止，我们已经实现了客户端发送数据和服务端打印，离本章开头提出的目标还差一半，接下来我们实现另外一半目标：服务端收到数据之后向客户端返回数据。

6.3 服务端返回数据到客户端

服务端向客户端写数据逻辑与客户端的写数据逻辑一样，首先创建一个ByteBuf，然后填充二进制数据，最后调用writeAndFlush()方法写出去。下面是服务端返回数据的代码。

```
public class FirstServerHandler extends ChannelInboundHandlerAdapter {

    @Override public void channelRead(ChannelHandlerContext ctx, Object msg) {

        // 接收数据逻辑省略

        // 返回数据到客户端

        System.out.println(new Date() + ": 服务端写出数据");

        ByteBuf out = getByteBuf(ctx);

        ctx.channel().writeAndFlush(out);

    }

    private ByteBuf getByteBuf(ChannelHandlerContext ctx) {

        byte [] bytes = "你好，欢迎关注我的微信公众号，《闪电侠的博客》！".getBytes(Charset.forName("utf-8"));

        ByteBuf buffer = ctx.alloc().buffer();

        buffer.writeBytes(bytes);

        return buffer;

    }

}
```

现在，轮到客户端了。客户端读取数据的逻辑和服务端读取数据的逻辑一样，同样是覆盖channelRead()方法。

```
public class FirstClientHandler extends ChannelInboundHandlerAdapter {  
  
    // 写数据相关的逻辑省略  
  
    @Override public void channelRead(ChannelHandlerContext ctx, Object msg) {  
  
        ByteBuf byteBuf = (ByteBuf) msg;  
  
        System.out.println(new Date() + ": 客户端读到数据 -> " +  
        byteBuf.toString(Charset.forName("utf-8")));  
  
    }  
  
}
```

将这段逻辑添加到客户端的逻辑处理器FirstClientHandler之后，客户端就能收到服务端发来的数据。

客户端与服务端读写数据的逻辑完成之后，先运行服务端，再运行客户端，控制台输出分别如下面两图所示。

服务端

```
端口[8000]绑定成功!  
Sat Aug 04 09:49:12 CST 2018: 服务端读到数据 -> 你好，闪电侠！  
Sat Aug 04 09:49:12 CST 2018: 服务端写出数据  
|
```

客户端

```
连接成功!  
Sat Aug 04 09:49:12 CST 2018: 客户端写出数据  
Sat Aug 04 09:49:12 CST 2018: 客户端读到数据 -> 你好，欢迎关注我的微信公众号，《闪电侠的博客》！  
|
```

到这里，本章要实现的客户端与服务端双向通信的功能就实现完毕了。

6.4 总结

- 首先，我们了解到客户端和服务端的逻辑处理均在启动的时候，通过为逻辑处理链 Pipeline添加逻辑处理器，来编写数据的读写逻辑。Pipeline的逻辑我们在后面会分析。
- 然后，在客户端连接成功之后，会回调到逻辑处理器的channelActive()方法。不管服务端还是客户端，收到数据之后都会调用channelRead()方法。

写数据调用writeAndFlush()方法，客户端与服务端交互的二进制数据载体为 ByteBuf，ByteBuf通过连接的内存管理器创建，字节数据填充到ByteBuf之后才能写到对端。接下来一章我们会重点分析ByteBuf。

6.5 思考

如何实现在新连接接入的时候，服务端主动向客户端推送消息，客户端回复服务端消息？

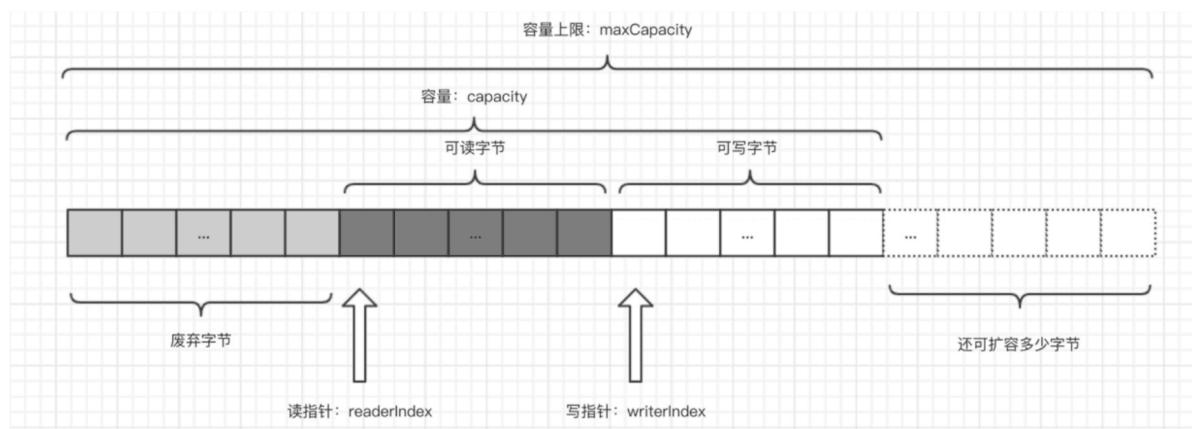
第7章

数据载体ByteBuf的介绍

在前面的章节中，我们已经了解到，Netty的数据读写是以ByteBuf为单位进行交互的。本章我们就来详细剖析一下ByteBuf。

7.1 ByteBuf的结构

首先，我们来了解一下ByteBuf的结构，如下图所示。



从ByteBuf的结构图可以看到：

1. ByteBuf是一个字节容器，容器里面的数据分为三部分，第一部分是已经丢弃的字节，这部分数据是无效的；第二部分是可读字节，这部分数据是ByteBuf的主体数据，从ByteBuf里读取的数据都来自这一部分；最后一部分的数据是可写字节，所有写到ByteBuf的数据都会写到这一段。后面的虚线部分表示该ByteBuf最多还能扩容多少容量。
2. 以上三部分内容是被两个指针划分出来的，从左到右依次是读指针 (readerIndex) 和写指针 (writerIndex)。还有一个变量capacity，表示ByteBuf底层内存的总容量。
3. 从ByteBuf中每读取一节，readerIndex自增1，ByteBuf里总共有writerIndex-readerIndex字节可读。由此可以知道，当readerIndex与writerIndex相等的时候，ByteBuf不可读。

4.写数据是从writerIndex指向的部分开始写的，每写一节，writerIndex自增1，直到增到capacity。这个时候，表示ByteBuf已经不可写。

5.ByteBuf里其实还有一个参数maxCapacity。当向ByteBuf写数据的时候，如果容量不足，则可以进行扩容，直到capacity扩容到maxCapacity，超过maxCapacity就会报错。

Netty使用ByteBuf这个数据结构可以有效地区分可读数据和可写数据，读写之间相互没有冲突。当然，ByteBuf只是对二进制数据的抽象，具体底层的实现我们后面会讲到。在这里，读者只需要知道Netty关于数据读写只认ByteBuf。下面我们来学习ByteBuf常用的API。

7.2 容量API

capacity()

表示ByteBuf底层占用了多少字节的内存（包括丢弃的字节、可读字节、可写字节），不同的底层实现机制有不同的计算方式，后面我们介绍ByteBuf的分类时会讲到。

maxCapacity()

表示ByteBuf底层最大能够占用多少字节的内存，当向ByteBuf中写数据的时候，如果发现容量不足，则进行扩容，直到扩容到maxCapacity，超过这个数，就抛出异常。

readableBytes()与isReadable()

readableBytes()表示ByteBuf当前可读的字节数，它的值等于writerIndex-readerIndex，如果两者相等，则不可读，isReadable()方法返回false。

writableBytes()、isWritable()与maxWritableBytes()

writableBytes()表示ByteBuf当前可写的字节数，它的值等于capacity-writerIndex，如果两者相等，则表示不可写，isWritable()返回false，但是这个时候，并不代表不能往ByteBuf写数据了。如果发现往ByteBuf写数据写不进去，Netty会自动扩容ByteBuf，直到底层的内存大小为maxCapacity，而maxWritableBytes()就表示可写的最大字节数，它的值等于maxCapacity-writerIndex。

7.3 读写指针相关的API

readerIndex()与readerIndex(int)

前者表示返回当前的读指针readerIndex，后者表示设置读指针。

writeIndex()与writeIndex(int)

前者表示返回当前的写指针writerIndex，后者表示设置写指针。

markReaderIndex()与resetReaderIndex()

前者表示把当前的读指针保存起来，后者表示把当前的读指针恢复到之前保存的值。下面两段代码是等价的。

// 代码片段一

```
int readerIndex = buffer.readerIndex();
```

// 其他操作

```
buffer.readerIndex(readerIndex);
```

// 代码片段二

```
buffer.markReaderIndex();
```

// 其他操作

```
buffer.resetReaderIndex();
```

希望大家多使用代码片段二这种方式，不需要自己定义变量。无论Buffer被当作参数传递到哪里，调用resetReaderIndex()都可以恢复到之前的状态，在解析自定义协议的数据包时非常常见，推荐大家使用这一对API。

markWriterIndex()与resetWriterIndex()

这一对API的作用与上一对API类似，这里不再赘述。

7.4 读写API

本质上，关于ByteBuf的读写都可以看作从指针开始的地方开始读写数据。

writeBytes(byte[] src)与buffer.readBytes(byte[] dst)

`writeBytes()`表示把字节数组src里的数据全部写到ByteBuf，而`readBytes()`表示把ByteBuf里的数据全部读取到dst。这里dst字节数组的大小通常等于`readableBytes()`，而src字节数组大小的长度通常小于等于`writableBytes()`。

writeByte(byte b)与buffer.readByte()

`writeByte()`表示往ByteBuf中写一字节，而`buffer.readByte()`表示从ByteBuf中读取一字节，类似的API还有`writeBoolean()`、`writeChar()`、`writeShort()`、`writeInt()`、`writeLong()`、`writeFloat()`、`writeDouble()`，以及`readBoolean()`、`readChar()`、`readShort()`、`readInt()`、`readLong()`、`readFloat()`、`readDouble()`，这里不再赘述，相信读者应该很容易理解这些API。

与读写API类似的API还有`getBytes()`、`getByte()`与`setBytes()`、`setByte()`系列，唯一的区别就是get、set不会改变读写指针，而read、write会改变读写指针，这一点在解析数据的时候千万要注意。

release()与retain()

由于Netty使用了堆外内存，而堆外内存是不被JVM直接管理的。也就是说，申请到的内存无法被垃圾回收器直接回收，所以需要我们手动回收。这有点类似于C语言里，申请到的内存必须手工释放，否则会造成内存泄漏。

Netty的ByteBuf是通过引用计数的方式管理的，如果一个ByteBuf没有地方被引用到，则需要回收底层内存。在默认情况下，当创建完一个ByteBuf时，它的引用为1，然后每次调用`retain()`方法，它的引用就加一，`release()`方法的原理是将引用计数减一，减完之后如果发现引用计数为0，则直接回收ByteBuf底层的内存。

slice()、duplicate()、copy()

在通常情况下，这三个方法会被放到一起比较，三者的返回值分别是一个新的ByteBuf对象。

1.`slice()`方法从原始ByteBuf中截取一段，这段数据是从`readerIndex`到`writeIndex`的，同时，返回的新的ByteBuf的最大容量`maxCapacity`为原始ByteBuf的`readableBytes()`。

2.`duplicate()`方法把整个ByteBuf都截取出来，包括所有的数据、指针信息。

3.`slice()`方法与`duplicate()`方法的相同点是：底层内存及引用计数与原始ByteBuf共享，也就是说，经过`slice()`方法或者`duplicate()`方法返回的ByteBuf调用`write`系列方法

都会影响到原始ByteBuf，但是它们都维持着与原始ByteBuf相同的内存引用计数和不同的读写指针。

4.slice()方法与duplicate()方法的不同点就是：slice()方法只截取从readerIndex到writerIndex之间的数据，它返回的ByteBuf的最大容量被限制到原始ByteBuf的readableBytes()，而duplicate()方法是把整个ByteBuf都与原始ByteBuf共享。

5.slice()方法与duplicate()方法不会复制数据，它们只是通过改变读写指针来改变读写的行为，而最后一个方法copy()会直接从原始ByteBuf中复制所有的信息，包括读写指针及底层对应的数据，因此，往copy()方法返回的ByteBuf中写数据不会影响原始ByteBuf。

6.slice()方法和duplicate()方法不会改变ByteBuf的引用计数，所以原始ByteBuf调用release()方法之后发现引用计数为零，就开始释放内存，调用这两个方法返回的ByteBuf也会被释放。这时候如果再对它们进行读写，就会报错。因此，我们可以通过调用一次retain()方法来增加引用，表示它们对应的底层内存多了一次引用，引用计数为2。在释放内存的时候，需要调用两次release()方法，将引用计数降到零，才会释放内存。

7.这三个方法均维护着自己的读写指针，与原始ByteBuf的读写指针无关，相互之间不受影响。

retainedSlice()与retainedDuplicate()

相信读者应该已经猜到这两个API的作用了，它们的作用是在截取内存片段的同时，增加内存的引用计数，分别与下面两段代码等价。

```
// retainedSlice 等价于  
  
slice().retain();  
  
// retainedDuplicate() 等价于  
  
duplicate().retain()
```

使用slice()和duplicate()方法的时候，千万要理清内存共享、引用计数共享、读写指针不共享等概念。下面举两个常见的容易出错的例子。

例1：多次释放

```
Buffer buffer = xxx;
```

```
doWith(buffer);

// 一次释放

buffer.release();

public void doWith(Bytebuf buffer) {

    // ...

    // 没有增加引用计数

    Buffer slice = buffer.slice();

    foo(slice);

}

public void foo(ByteBuf buffer) {

    // 从缓冲区读取并处理

    // 重复释放

    buffer.release();

}
```

这里的doWith有时候是用户自定义的方法，有时候是Netty的回调方法，如channelRead()等。

例2：不释放造成内存泄漏

```
Buffer buffer = xxx;

doWith(buffer);

// 引用计数为2，调用release()方法之后，引用计数为1，无法释放内存

buffer.release();

public void doWith(Bytebuf buffer) {
```

```
// ...

// 增加引用计数

Buffer slice = buffer.retainSlice();

foo(slice);

// 没有调用release()方法

}

public void foo(ByteBuf buffer) {

    //从缓冲区读取并处理

}
```

想要避免以上两种情况的发生，大家只需要记住一点，在一个函数体里面，只要增加了引用计数（包括ByteBuf的创建和手动调用retain()方法），就必须调用release()方法。

7.5 实战

了解了以上API之后，我们使用上述API来写一个简单的Demo。

ByteBufTest.java

```
public class ByteBufTest {

    public static void main(String [] args) {

        ByteBuf buffer = ByteBufAllocator.DEFAULT.buffer(9, 100);

        print("allocate ByteBuf(9, 100)", buffer);

        // write方法改变写指针，写完之后写指针未到capacity的时候，buffer仍然可写

        buffer.writeBytes(new byte [] {1, 2, 3, 4});

        print("writeBytes(1,2,3,4)", buffer);
```

```
// write方法改变写指针，写完之后写指针未到capacity的时候，buffer仍然可写，写完
int 类型之后，写指针增加4

buffer.writeInt(12);

print("writeInt(12)", buffer);

// write方法改变写指针，写完之后写指针等于capacity的时候，buffer不可写

buffer.writeBytes(new byte [] {5} );

print("writeBytes(5)", buffer);

// write方法改变写指针，写的时候发现buffer不可写则开始扩容，扩容之后capacity随即
改变

buffer.writeBytes(new byte [] {6} );

print("writeBytes(6)", buffer);

// get方法不改变读写指针

System.out.println("getByte(3) return: " + buffer.getByte(3));

System.out.println("getShort(3) return: " + buffer.getShort(3));

System.out.println("getInt(3) return: " + buffer.getInt(3));

print("getByte()", buffer);

// set方法不改变读写指针

buffer.setByte(buffer.readableBytes() + 1, 0);

print("setByte()", buffer);

// read方法改变读指针

byte [] dst = new byte [buffer.readableBytes()] ;

buffer.readBytes(dst);
```

```
    print("readBytes(" + dst.length + ")", buffer);

}

private static void print(String action, ByteBuf buffer)  {

    System.out.println("after =====allocate ByteBuf(9, 100)=====");

    System.out.println("capacity(): " + buffer.capacity());

    System.out.println("maxCapacity(): " + buffer.maxCapacity());

    System.out.println("readerIndex(): " + buffer.readerIndex());

    System.out.println("readableBytes(): " + buffer.readableBytes());

    System.out.println("isReadable(): " + buffer.isReadable());

    System.out.println("writerIndex(): " + buffer.writerIndex());

    System.out.println("writableBytes(): " + buffer.writableBytes());

    System.out.println("isWritable(): " + buffer.isWritable());

    System.out.println("maxWritableBytes(): " +
buffer.maxWritableBytes());

    System.out.println();

}

}
```

控制台输出如下。

```
after =====allocate ByteBuf(9, 100)=====

capacity(): 9

maxCapacity(): 100

readerIndex(): 0
```

```
readableBytes(): 0
isReadable(): false writerIndex(): 0
writableBytes(): 9
isWritable(): true maxWritableBytes(): 100
after =====writeBytes(1, 2, 3, 4)=====
capacity(): 9
maxCapacity(): 100
readerIndex(): 0
readableBytes(): 4
isReadable(): true writerIndex(): 4
writableBytes(): 5
isWritable(): true maxWritableBytes(): 96
after =====writeInt(12)=====
capacity(): 9
maxCapacity(): 100
readerIndex(): 0
readableBytes(): 8
isReadable(): true writerIndex(): 8
writableBytes(): 1
isWritable(): true maxWritableBytes(): 92
after =====writeBytes(5)=====
```

```
capacity(): 9
maxCapacity(): 100
readerIndex(): 0
readableBytes(): 9
isReadable(): true writerIndex(): 9
writableBytes(): 0
isWritable(): false maxWritableBytes(): 91
after =====writeBytes(6)=====
capacity(): 64
maxCapacity(): 100
readerIndex(): 0
readableBytes(): 10
isReadable(): true writerIndex(): 10
writableBytes(): 54
isWritable(): true maxWritableBytes(): 90
getByte(3) return: 4
getShort(3) return: 1024
getInt(3) return: 67108864
after =====getByte()=====
capacity(): 64
maxCapacity(): 100
```

```
readerIndex(): 0
readableBytes(): 10
isReadable(): true writerIndex(): 10
writableBytes(): 54
isWritable(): true maxWritableBytes(): 90
after =====setByte()=====
capacity(): 64
maxCapacity(): 100
readerIndex(): 0
readableBytes(): 10
isReadable(): true writerIndex(): 10
writableBytes(): 54
isWritable(): true maxWritableBytes(): 90
after =====readBytes(10)=====
capacity(): 64
maxCapacity(): 100
readerIndex(): 10
readableBytes(): 0
isReadable(): false writerIndex(): 10
writableBytes(): 54
isWritable(): true maxWritableBytes(): 90
```

相信读者在了解了ByteBuf的结构之后，不难理解控制台的输出。读者可以自己花时间分析一下控制台的输出。

7.6 总结

- 1.本章我们分析了Netty对二进制数据的抽象ByteBuf的结构，本质上它的原理就是，引用了一段内存，这段内存可以是堆内的，也可以是堆外的，然后用引用计数来控制这段内存是否需要被释放。使用读写指针来控制ByteBuf的读写，可以理解为是外观模式的一种使用。
- 2.基于读写指针和容量、最大可扩容容量，衍生出一系列读写方法，要注意read、write与get、set的区别。
- 3.多个ByteBuf可以引用同一段内存，通过引用计数来控制内存的释放，遵循谁retain()谁release()的原则。
- 4.最后，我们通过一个具体的例子说明了ByteBuf的实际使用。

7.7 思考

slice()方法可能用在什么场景？欢迎加入本书读者群进行讨论。

第8章

客户端与服务端通信协议编解码

在学习了ByteBuf的API之后，本章我们来学习如何设计并实现客户端与服务端的通信协议。

8.1 什么是客户端与服务端的通信协议

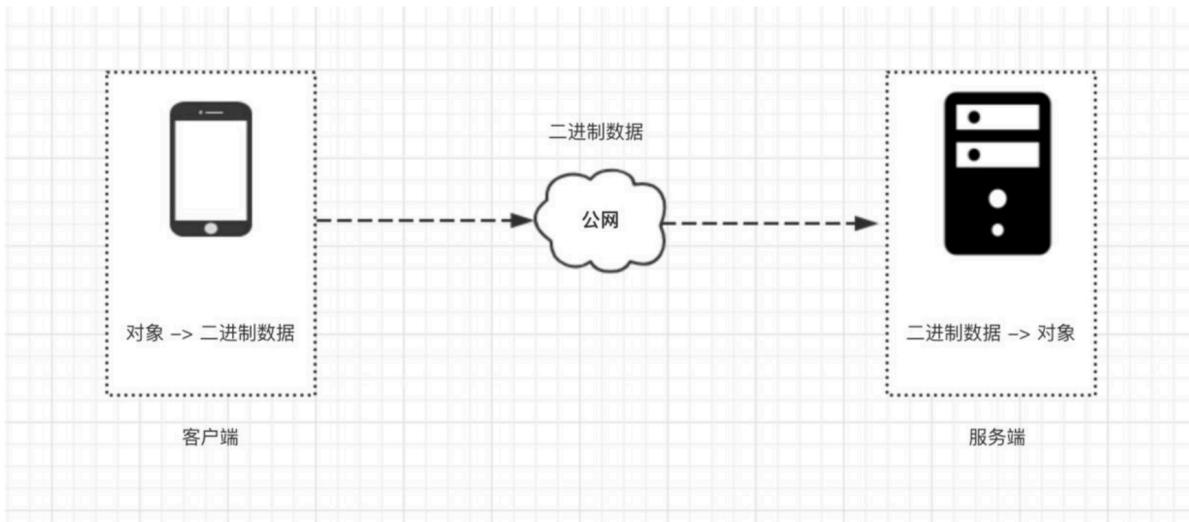
无论使用Netty还是使用原始的Socket编程，基于TCP通信的数据包格式均为二进制，协议指的就是客户端与服务端事先商量好的，每一个二进制数据包中的每一段字节分别代表什么含义的规则。一个简单的登录指令如下图所示。

| 指令 | 用户名 | | | | | 分隔符 | 密码 | | |
|----|-----|-----|-----|-----|-----|------|-----|-----|-----|
| 1 | 'f' | 'l' | 'a' | 's' | 'h' | '\0' | 'p' | 'w' | 'd' |

在这个数据包中，第一个字节为1表示这是一个登录指令，接下来是用户名和密码，这两个值以\0分割，客户端发送这段二进制数据包到服务端，服务端就能根据这个协议来取出用户名和密码，执行登录逻辑。在实际的通信协议设计中，我们会考虑更多细节，比这个协议稍微复杂一些。

那么，协议设计好之后，客户端与服务端的通信过程又是怎样的呢？

客户端与服务端的通信如下图所示。



1. 客户端把一个Java对象按照通信协议转换成二进制数据包。
2. 通过网络，把这段二进制数据包发送到服务端。在数据的传输过程中，由TCP/IP协议负责数据的传输，与应用层无关。
3. 服务端接收到数据之后，按照协议取出二进制数据包中的相应字段，包装成Java对象，交给应用逻辑处理。
4. 服务端处理完之后，如果需要生成响应给客户端，则按照相同的流程进行。

第1章已经列出了实现一个支持单聊和群聊的IM指令集合，设计协议的目的就是客户端与服务端能够识别这些具体的指令。接下来，我们就看一下如何设计这个通信协议。

8.2 通信协议的设计

通信协议的设计如下图所示。



1. 第一个字段是魔数，通常情况下为固定的几字节（这里规定为4字节）。为什么需要这个字段，而且还是一个固定的数？假设我们在服务器上开了一个端口，比如80端

口，如果没有这个魔数，任何数据包传递到服务器，服务器都会根据自定义协议来进行处理，包括不符合自定义协议规范的数据包。例如，直接通过http://服务器IP来访问服务器（默认为80端口），服务端收到的是一个标准的HTTP协议数据包，但是它仍然会按照事先约定好的协议来处理HTTP协议，显然，这是会解析出错的。而有了这个魔数之后，服务端首先取出前面4字节进行比对，能够在第一时间识别出这个数据包并非是遵循自定义协议的，也就是无效数据包，出于安全考虑，可以直接关闭连接以节省资源。在Java字节码的二进制文件中，开头的4字节为0xcafebabe，用来标识这是一个字节码文件，亦有异曲同工之妙。

2.接下来的1字节为版本号，通常情况下是预留字段，在协议升级的时候用到，有点类似TCP协议中的一个字段标识是IPV4协议还是IPV6协议。在大多数情况下，这个字段是用不到的，但是为了协议能够支持升级，还是先留着。

3.第三部分的序列化算法表示如何把Java对象转换为二进制数据及二进制数据如何转换回Java对象，比如Java自带的序列化、JSON、Hessian等序列化方式。

4.第四部分的字段表示指令，关于指令相关的介绍，我们在前面已经讨论过。服务端或者客户端每收到一种指令，都会有相应的处理逻辑。这里我们用1字节来表示，最高支持256种指令，对于这个即时聊天系统来说已经完全足够了。

5.第五部分的字段表示数据长度，占4字节。

6.最后一部分为数据内容，每一种指令对应的数据都是不一样的，比如登录的时候需要用户名和密码，收消息的时候需要用户标识和具体消息内容等。

在通常情况下，这样一套标准的协议能够适配大多数情况下的客户端与服务端的通信场景，接下来我们就看一下如何使用Netty来实现这套协议。

8.3 通信协议的实现

我们把Java对象根据协议封装成二进制数据包的过程称为编码，把从二进制数据包中解析出Java对象的过程称为解码。在学习如何使用Netty进行通信协议的编解码之前，我们先来定义一下客户端与服务端通信的Java对象。

8.3.1 Java对象

如下代码定义通信过程中的Java对象。

```
@Data
```

```
public abstract class Packet {  
    /**  
     * 协议版本  
     */  
  
    private Byte version = 1;  
    /**  
     * 指令  
     */  
  
    public abstract Byte getCommand();  
}
```

1.以上是通信过程中Java对象的抽象类。可以看到，我们定义了一个版本号（默认值为1），以及一个获取指令的抽象方法。所有的指令数据包都必须实现这个方法，这样我们就可以知道某种指令的含义。

2.@Data注解由lombok提供，它会自动帮我们生产getter、setter方法，减少大量重复代码，推荐使用。

接下来，以客户端登录请求为例，定义登录请求数据包。

```
public interface Command {  
    Byte LOGIN_REQUEST = 1;  
}  
  
@Data  
  
public class LoginRequestPacket extends Packet {  
    private Integer userId;  
    private String username;
```

```
private String password;

@Override

public Byte getCommand()  {

    return LOGIN_REQUEST;

}

}
```

登录请求数据包继承自Packet定义了3个字段，分别是用户ID、用户名和密码。其中最为重要的就是覆盖了父类的getCommand()方法，值为常量LOGIN_REQUEST。

Java对象定义完成之后，我们需要定义一种规则，如何把一个Java对象转换成二进制数据，这个规则叫作Java对象的序列化。

8.3.2 序列化

如下代码定义序列化接口。

```
public interface Serializer  {

    /**
     * 序列化算法
     */

    byte getSerializerAlgorithm();

    /**
     * Java 对象转换成二进制数据
     */

    byte [] serialize(Object object);

    /**

```

* 二进制数据转换成Java对象

*/

```
<T> T deserialize(Class<T> clazz, byte [] bytes);  
}
```

序列化接口有3个方法：getSerializerAlgorithm()方法获取具体的序列化算法标识，serialize()方法将Java对象转换成字节数组，deserialize()方法将字节数组转换成某种类型的Java对象。在本书中，我们使用最简单的JSON序列化方式，将阿里巴巴的Fastjson作为序列化框架。

```
public interface SerializerAlgorithm {  
    /**  
     * JSON序列化标识  
     */  
    byte JSON = 1;  
}  
  
public class JSONSerializer implements Serializer {  
    @Override  
    public byte getSerializerAlgorithm() {  
        return SerializerAlgorithm.JSON;  
    }  
  
    @Override  
    public byte [] serialize(Object object) {  
        return JSON.toJSONString(object);  
    }  
}
```

```
    @Override

    public <T> T deserialize(Class<T> clazz, byte [] bytes) {
        return JSON.parseObject(bytes, clazz);
    }
}
```

然后，我们定义一下序列化算法的类型，以及默认序列化算法。

```
public interface Serializer {
    /**
     * JSON序列化
     */
    byte JSON_SERIALIZER = 1;

    Serializer DEFAULT = new JSONSerializer();

    // ...
}
```

这样，我们就实现了序列化相关的逻辑。如果想要实现其他序列化算法，则只需要继承Serializer，然后定义序列化算法的标识，再覆盖两个方法即可。

序列化定义了Java对象与二进制数据的互转过程。接下来，我们学习如何把这部分数据编码到通信协议的二进制数据包中去。

8.3.3 编码：封装成二进制数据的过程

PacketCodeC.java

```
private static final int MAGIC_NUMBER = 0x12345678;

public ByteBuf encode(Packet packet) {
```

```
// 1. 创建 ByteBuf 对象

ByteBuf byteBuf = ByteBufAllocator.DEFAULT.ioBuffer();

// 2. 序列化 Java 对象

byte [] bytes = Serializer.DEFAULT.serialize(packet);

// 3. 实际编码过程

byteBuf.writeInt(MAGIC_NUMBER);

byteBuf.writeByte(packet.getVersion());

byteBuf.writeByte(Serializer.DEFAULT.getSerializerAlgorithm());

byteBuf.writeByte(packet.getCommand());

byteBuf.writeInt(bytes.length);

byteBuf.writeBytes(bytes);

return byteBuf;

}
```

编码过程分为3个步骤。

- 1.我们需要创建一个ByteBuf，这里我们调用Netty的ByteBuf分配器来创建，`ioBuffer()`方法会返回适配IO读写相关的内存，它会尽可能创建一个直接内存。直接内存可以理解为不受JVM堆管理的内存空间，写到IO缓冲区的效果更高。
- 2.将Java对象序列化成二进制数据包。
- 3.我们对照本章开头的协议设计和上一章ByteBuf的API，逐个往ByteBuf写入字段，即实现了编码过程。到此，编码过程结束。

一端实现编码之后，Netty会将此ByteBuf写到另一端。另一端获得的也是一个ByteBuf对象。基于这个ByteBuf对象，就可以反解出在对端创建的Java对象，这个过程被称作解码，下面我们就来分析这个过程。

8.3.4 解码：解析Java对象的过程

PacketCodeC.java

```
public Packet decode(ByteBuf byteBuf) {  
  
    // 跳过魔数  
  
    byteBuf.skipBytes(4);  
  
    // 跳过版本号  
  
    byteBuf.skipBytes(1);  
  
    // 序列化算法标识  
  
    byte serializeAlgorithm = byteBuf.readByte();  
  
    // 指令  
  
    byte command = byteBuf.readByte();  
  
    // 数据包长度  
  
    int length = byteBuf.readInt();  
  
    byte [] bytes = new byte [length] ;  
  
    byteBuf.readBytes(bytes);  
  
    Class<? extends Packet> requestType = getRequestType(command);  
  
    Serializer serializer = getSerializer(serializeAlgorithm);  
  
    if (requestType != null && serializer != null) {  
  
        return serializer.deserialize(requestType, bytes);  
  
    }  
  
    return null;  
  
}
```

解码的流程如下。

1. 我们假定decode方法传递进来的ByteBuf已经是合法的（后面我们再来实现校验），即首个4字节是我们定义的魔数0x12345678，这里我们调用skipBytes跳过这个4字节。
2. 我们暂时不关注协议版本，通常在没有遇到协议升级的时候，暂时不处理这个字段。因为在绝大多数情况下，几乎用不着这个字段，但仍然需要暂时保留。
3. 我们调用ByteBuf的API分别获得序列化算法标识、指令、数据包的长度。
4. 我们根据获得的数据包的长度取出数据，通过指令获得该数据包对应的Java对象的类型，根据序列化算法标识获得序列化对象，将字节数组转换为Java对象。至此，解码过程结束。

由此可以看到，解码过程与编码过程正好是相反的过程。

8.4 总结

本章，我们学到了以下几个知识点。

1. 通信协议是为了客户端与服务端交互，双方协商出来的满足一定规则的二进制数据格式。
2. 介绍了一种通用的通信协议的设计，包括魔数、版本号、序列化算法标识、指令、数据长度、数据几个字段，该协议能够满足绝大多数通信场景。
3. Java对象及序列化的目的就是实现Java对象与二进制数据的互转。
4. 我们依照设计的协议和ByteBuf的API实现了通信协议，这个过程被称为编解码过程。

8.5 思考

1. 除了JSON序列化方式，还有哪些序列化方式？如何实现？
2. 序列化和编码都是把Java对象封装成二进制数据的过程，这两者有什么区别和联系？

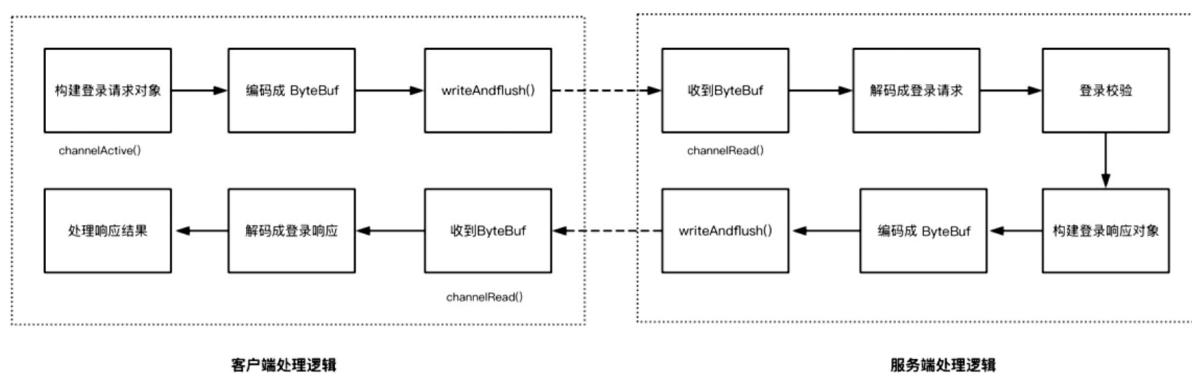
第9章

实现客户端登录

本章我们来实现客户端登录到服务端的过程。

9.1 登录流程

登录流程如下图所示。



由上图可以看到，客户端连接上服务端之后：

- 1.客户端首先会构建一个登录请求对象，然后通过编码把请求对象编码为ByteBuf，写到服务端。
- 2.服务端接收到ByteBuf之后，首先通过解码把ByteBuf解码为登录请求响应，然后进行校验。
- 3.服务端校验通过之后，构造一个登录响应对象，依然经过编码，再写回客户端。
- 4.客户端接收服务端的响应数据之后，解码ByteBuf，获得登录响应对象，判断是否登录成功。

9.2 逻辑处理器

下面来分别实现上述4个过程。开始之前，我们先回顾一下客户端与服务端的启动流程，当客户端启动的时候，我们在引导类Bootstrap中配置客户端的处理逻辑。本节中我们给客户端配置的逻辑处理器被叫作ClientHandler。

```
public class ClientHandler extends ChannelInboundHandlerAdapter {  
    }  
}
```

然后，在客户端启动的时候，我们给Bootstrap配置上这个逻辑处理器。

```
bootstrap.handler(new ChannelInitializer<SocketChannel>() {  
    @Override public void initChannel(SocketChannel ch) {  
        ch.pipeline().addLast(new ClientHandler());  
    }  
});
```

这样，在客户端Netty中IO事件相关的回调就能够回调到ClientHandler。

同样，我们给服务端引导类ServerBootstrap也配置一个逻辑处理器：
ServerHandler。

```
public class ServerHandler extends ChannelInboundHandlerAdapter {  
    }  
  
serverBootstrap.childHandler(new ChannelInitializer<NioSocketChannel>()  
    {  
        protected void initChannel(NioSocketChannel ch) {  
            ch.pipeline().addLast(new ServerHandler());  
        }  
    }  
}
```

这样，在服务端Netty中IO事件相关的回调就能够回调到ServerHandler。

接下来，我们就围绕这两个Handler编写处理逻辑。

9.3 客户端发送登录请求

9.3.1 客户端处理登录请求

我们实现客户端与服务端连接之后，立即登录。在连接上服务端之后，Netty会回调 ClientHandler的channelActive()方法，我们在这个方法体里编写相应的逻辑。

ClientHandler.java

```
public void channelActive(ChannelHandlerContext ctx) {  
  
    System.out.println(new Date() + ": 客户端开始登录");  
  
    // 创建登录对象  
  
    LoginRequestPacket loginRequestPacket = new LoginRequestPacket();  
  
    loginRequestPacket.setUserId(UUID.randomUUID().toString());  
  
    loginRequestPacket.setUsername("flash");  
  
    loginRequestPacket.setPassword("pwd");  
  
    // 编码  
  
    ByteBuf buffer = PacketCodeC.INSTANCE.encode(ctx.alloc(),  
        loginRequestPacket);  
  
    // 写数据  
  
    ctx.channel().writeAndFlush(buffer);  
  
}
```

在编码环节，我们把PacketCodeC变成单例模式，然后从ByteBuf分配器抽取出一个参数，这里第一个实参ctx.alloc()获取的就是与当前连接相关的ByteBuf分配器，建议这样使用。

写数据的时候，我们首先通过ctx.channel()获取当前连接（Netty对连接的抽象为 Channel，后面章节会分析），然后调用writeAndFlush()把二进制数据写到服务端。这样，客户端发送登录请求的逻辑就完成了。接下来，我们介绍服务端接收到这个数据之后是如何处理的。

9.3.2 服务端处理登录请求

ServerHandler.java

```
public void channelRead(ChannelHandlerContext ctx, Object msg) {  
  
    ByteBuf requestByteBuf = (ByteBuf) msg;  
  
    // 解码  
  
    Packet packet = PacketCodeC.INSTANCE.decode(requestByteBuf);  
  
    // 判断是否是登录请求数据包  
  
    if (packet instanceof LoginRequestPacket) {  
  
        LoginRequestPacket loginRequestPacket = (LoginRequestPacket) packet;  
  
        // 登录校验  
  
        if (valid(loginRequestPacket)) {  
  
            // 校验成功  
  
        } else {  
  
            // 校验失败  
  
        }  
  
    }  
  
}  
  
private boolean valid(LoginRequestPacket loginRequestPacket) {  
  
    return true;  
  
}
```

向服务端引导类ServerBootstrap中添加逻辑处理器ServerHandler，Netty在收到数据之后，会回调channelRead()方法。这里的第二个参数msg，在这个场景中，可以直

接强转为ByteBuf。为什么Netty不直接把这个参数类型定义为ByteBuf？我们在后续的内容中会分析。

拿到ByteBuf之后，首先要做的事情就是解码，解码出Java数据包对象，然后判断如果是登录请求数据包LoginRequestPacket，就进行登录逻辑的处理。这里，我们假设所有的登录都是成功的，valid()方法返回true。

服务端校验通过之后，接下来就需要向客户端发送登录响应，我们继续编写服务端的逻辑。

9.4 服务端发送登录响应

9.4.1 服务端处理登录响应

ServerHandler.java

```
LoginResponsePacket loginResponsePacket = new LoginResponsePacket();

loginResponsePacket.setVersion(packet.getVersion());

if (valid(loginRequestPacket)) {
    loginResponsePacket.setSuccess(true);

} else {

    loginResponsePacket.setReason("账号密码校验失败");

    loginResponsePacket.setSuccess(false);

}

// 编码

ByteBuf responseByteBuf = PacketCodeC.INSTANCE.encode(ctx.alloc(),
loginResponsePacket);

ctx.channel().writeAndFlush(responseByteBuf);
```

这段代码仍然是在服务端逻辑处理器ServerHandler的channelRead()方法里构造一个登录响应包LoginResponsePacket，然后在校验成功和失败的时候分别设置标志位，

接下来调用编码器把Java对象编码成ByteBuf，调用writeAndFlush()写到客户端。至此，服务端的登录逻辑编写完成。还有最后一步，即客户端处理登录响应。

9.4.2 客户端处理登录响应

客户端接收服务端数据的处理逻辑也在ClientHandler的channelRead()方法中。

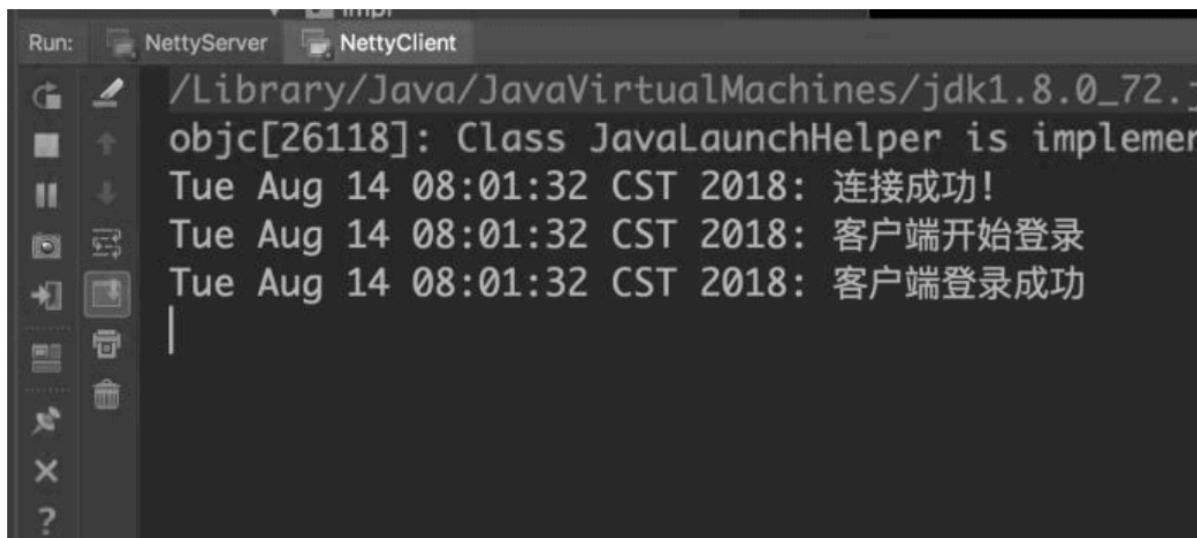
```
public void channelRead(ChannelHandlerContext ctx, Object msg) {  
  
    ByteBuf byteBuf = (ByteBuf) msg;  
  
    Packet packet = PacketCodeC.INSTANCE.decode(byteBuf);  
  
    if (packet instanceof LoginResponsePacket) {  
  
        LoginResponsePacket loginResponsePacket = (LoginResponsePacket)  
packet;  
  
        if (loginResponsePacket.isSuccess()) {  
  
            System.out.println(new Date() + ": 客户端登录成功");  
  
        } else {  
  
            System.out.println(new Date() + ": 客户端登录失败, 原因: " +  
loginResponsePacket.getReason());  
  
        }  
    }  
}
```

客户端拿到数据之后，调用PacketCodeC进行解码操作，如果类型是登录响应数据包，则逻辑比较简单，在控制台打印出一条消息。

至此，客户端整个登录流程就结束了。这里为了给大家演示，客户端和服务端的处理逻辑都较为简单，但是相信大家应该已经掌握了使用Netty来进行客户端与服务端交互的基本思路。基于这个思路，再运用到实际项目中，并不是难事。

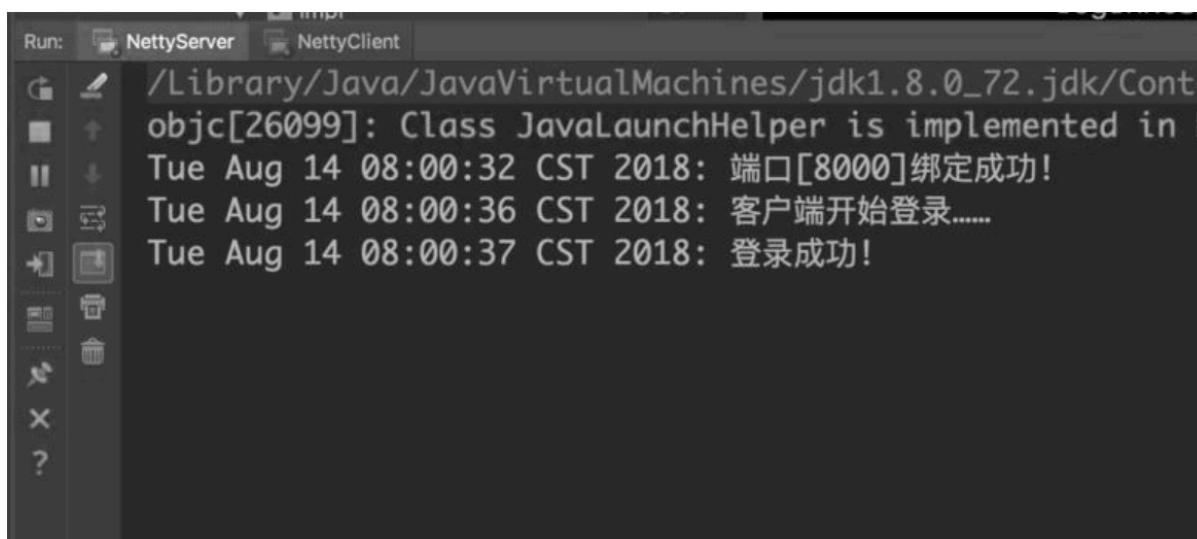
最后，我们来看一下效果，下面两图分别是客户端与服务端的控制台输出。

客户端



```
Run: NettyServer NettyClient
/Library/Java/JavaVirtualMachines/jdk1.8.0_72.jdk/Cont
objc[26118]: Class JavaLaunchHelper is implemented in
Tue Aug 14 08:01:32 CST 2018: 连接成功!
Tue Aug 14 08:01:32 CST 2018: 客户端开始登录
Tue Aug 14 08:01:32 CST 2018: 客户端登录成功
```

服务端



```
Run: NettyServer NettyClient
/Library/Java/JavaVirtualMachines/jdk1.8.0_72.jdk/Cont
objc[26099]: Class JavaLaunchHelper is implemented in
Tue Aug 14 08:00:32 CST 2018: 端口[8000]绑定成功!
Tue Aug 14 08:00:36 CST 2018: 客户端开始登录.....
Tue Aug 14 08:00:37 CST 2018: 登录成功!
```

9.5 总结

本章我们梳理了客户端登录的基本流程，然后结合上一章的编解码逻辑，使用Netty实现了完整的客户端登录流程。

9.6 思考

客户端登录成功或者失败之后，如何把成功或者失败的标识绑定在客户端的连接上？
服务端又如何高效避免客户端重新登录？

第10章

实现客户端与服务端收发消息

这一章，我们来实现客户端与服务端收发消息，要实现的具体功能是：在控制台输入一条消息之后按回车键，校验完客户端的登录状态之后，把消息发送到服务端；服务端收到消息之后打印，并向客户端发送一条消息，客户端收到消息之后打印。

10.1 收发消息对象

首先，我们来定义一下客户端与服务端的收发消息对象。我们把客户端发送至服务端的消息对象定义为MessageRequestPacket。

```
@Data  
  
public class MessageRequestPacket extends Packet {  
  
    private String message;  
  
    @Override  
  
    public Byte getCommand() {  
  
        return MESSAGE_REQUEST;  
  
    }  
  
}
```

指令为MESSAGE_REQUEST = 3。

我们把服务端发送至客户端的消息对象定义为MessageResponsePacket。

```
@Data  
  
public class MessageResponsePacket extends Packet {  
  
    private String message;
```

```
@Override  
  
public Byte getCommand() {  
  
    return MESSAGE_RESPONSE;  
  
}  
  
}
```

指令为MESSAGE_RESPONSE=4。

至此，我们的指令已经有如下4种。

```
public interface Command {  
  
    Byte LOGIN_REQUEST = 1;  
  
    Byte LOGIN_RESPONSE = 2;  
  
    Byte MESSAGE_REQUEST = 3;  
  
    Byte MESSAGE_RESPONSE = 4;  
  
}
```

10.2 判断客户端是否登录成功

在第9章中，我们出了一道思考题：如何判断客户端是否已经登录？

在客户端启动流程这一章，我们提到可以给客户端连接也就是Channel绑定属性，那么通过channel.attr(xxx).set(xx)方式，是否可以在登录成功之后，给Channel绑定一个登录成功的标志位，然后在判断是否登录成功的时候取出这个标志位呢？答案是肯定的。

首先定义登录成功的标志位。

```
public interface Attributes {
```

```
AttributeKey<Boolean> LOGIN = AttributeKey.newInstance("login");

    }
```

然后在客户端登录成功之后，给客户端绑定登录成功的标志位。

ClientHandler.java

```
public void channelRead(ChannelHandlerContext ctx, Object msg) {

    // ...

    if (loginResponsePacket.isSuccess()) {

        LoginUtil.markAsLogin(ctx.channel());

        System.out.println(new Date() + ": 客户端登录成功");

    } else {

        System.out.println(new Date() + ": 客户端登录失败, 原因: " +
            loginResponsePacket.getReason());

    }

    // ...
}
```

这里，我们省去了非关键代码部分。

```
public class LoginUtil {

    public static void markAsLogin(Channel channel) {

        channel.attr(Attributes.LOGIN).set(true);

    }

    public static boolean hasLogin(Channel channel) {
```

```
Attribute<Boolean> loginAttr = channel.attr(Attributes.LOGIN);

return loginAttr.get() != null;

}

}
```

如上代码所示，我们抽取出LoginUtil用于设置登录标志位并判断是否有标志位。如果有标志位，不管标志位的值是什么，都表示已经成功登录。接下来，我们实现控制台输入消息并发送至服务端。

10.3 在控制台输入消息并发送

现在，我们在客户端连接上服务端之后启动控制台线程，从控制台获取消息，然后发送至服务端。

NettyClient.java

```
private static void connect(Bootstrap bootstrap, String host, int
port, int retry)  {

bootstrap.connect(host, port).addListener(future ->  {

if (future.isSuccess())  {

Channel channel = ((ChannelFuture) future).channel();

// 连接成功之后，启动控制台线程

startConsoleThread(channel);

}

// ...

});

}

private static void startConsoleThread(Channel channel)  {
```

```
new Thread(() -> {

    while (! Thread.interrupted()) {

        if (LoginUtil.hasLogin(channel)) {

            System.out.println("输入消息发送至服务端: ");

            Scanner sc = new Scanner(System.in);

            String line = sc.nextLine();

            MessageRequestPacket packet = new MessageRequestPacket();

            packet.setMessage(line);

            ByteBuf byteBuf = PacketCodeC.INSTANCE.encode(channel.alloc(),
                packet);

            channel.writeAndFlush(byteBuf);

        }

    }

} ).start();

}
```

这里，我们省略了非关键代码。连接成功之后，调用startConsoleThread()开始启动控制台线程。在控制台线程中，判断只要当前Channel是登录状态，就允许控制台输入消息。

从控制台获取消息之后，将消息封装成消息对象，然后将消息编码成ByteBuf，最后通过writeAndFlush()将消息写到服务端，这个过程相信大家在学习了上节内容后，应该不会太陌生。接下来，我们介绍服务端收到消息之后是如何处理的。

10.4 服务端收发消息处理

ServerHandler.java

```
public void channelRead(ChannelHandlerContext ctx, Object msg) {  
  
    ByteBuf requestByteBuf = (ByteBuf) msg;  
  
    Packet packet = PacketCodeC.INSTANCE.decode(requestByteBuf);  
  
    if (packet instanceof LoginRequestPacket) {  
  
        // 处理登录  
  
    } else if (packet instanceof MessageRequestPacket) {  
  
        // 处理消息  
  
        MessageRequestPacket messageRequestPacket = ((MessageRequestPacket) packet);  
  
        System.out.println(new Date() + ": 收到客户端消息: " +  
  
messageRequestPacket.getMessage());  
  
        MessageResponsePacket messageResponsePacket = new  
MessageResponsePacket();  
  
        messageResponsePacket.setMessage("服务端回复【" +  
  
messageRequestPacket.getMessage() + "】");  
  
        ByteBuf responseByteBuf = PacketCodeC.INSTANCE.encode(ctx.alloc(),  
messageResponsePacket);  
  
        ctx.channel().writeAndFlush(responseByteBuf);  
  
    }  
}
```

服务端在收到消息之后，仍然回调到channelRead()方法，解码之后用一个else分支进入消息处理的流程。

首先服务端将收到的消息打印到控制台，然后封装一个消息响应对象 MessageResponsePacket，接下来编码成ByteBuf，再调用writeAndFlush()将数据写到客户端。我们再来看一下客户端收到消息的逻辑。

10.5 客户端收消息处理

ClientHandler.java

```
public void channelRead(ChannelHandlerContext ctx, Object msg) {  
  
    ByteBuf byteBuf = (ByteBuf) msg;  
  
    Packet packet = PacketCodeC.INSTANCE.decode(byteBuf);  
  
    if (packet instanceof LoginResponsePacket) {  
  
        // 登录逻辑  
  
    } else if (packet instanceof MessageResponsePacket) {  
  
        MessageResponsePacket messageResponsePacket =  
(MessageResponsePacket) packet;  
  
        System.out.println(new Date() + ": 收到服务端的消息: " +  
  
messageResponsePacket.getMessage());  
  
    }  
  
}
```

客户端在收到消息之后，回调到channelRead()方法，仍然用一个else逻辑进入消息处理的逻辑，这里我们仅仅简单地打印出消息。

最后，我们来看一下客户端和服务端的运行效果图。

客户端

```
Debug: NettyServer NettyClient
Debugger Console
/Library/Java/JavaVirtualMachines/jdk1.8.0_72.jdk/Contents/Home/bin/java ...
objc[28902]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines/jdk1.8.0_72.jdk/Contents/Home/bin/java ...
Connected to the target VM, address: '127.0.0.1:62412', transport: 'socket'
Wed Aug 15 08:05:53 CST 2018: 连接成功, 启动控制台线程.....
Wed Aug 15 08:05:53 CST 2018: 客户端开始登录
Wed Aug 15 08:05:53 CST 2018: 客户端登录成功
输入消息发送至服务端:
你好。你的微信公众号《闪电侠的博客》里的小闪对话系列博客太精彩了!
输入消息发送至服务端:
Wed Aug 15 08:07:10 CST 2018: 收到服务端的消息: 服务端回复【你好，你的微信公众号《闪电侠的博客》里的小闪对话系列博客太精彩了！】
感谢你用微信对话这种非常通俗的形式来讲技术，别断更啊!
输入消息发送至服务端:
Wed Aug 15 08:08:20 CST 2018: 收到服务端的消息: 服务端回复【感谢你用微信对话这种非常通俗的形式来讲技术，别断更啊！】
```

服务端

```
Debug: NettyServer NettyClient
Debugger Console
/Library/Java/JavaVirtualMachines/jdk1.8.0_72.jdk/Contents/Home/bin/java ...
objc[28900]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines/jdk1.8.0_72.jdk/Contents/Home/bin/java ...
Connected to the target VM, address: '127.0.0.1:62404', transport: 'socket'
Wed Aug 15 08:05:48 CST 2018: 端口[8000]绑定成功!
Wed Aug 15 08:05:53 CST 2018: 收到客户端登录请求.....
Wed Aug 15 08:05:53 CST 2018: 登录成功!
Wed Aug 15 08:07:10 CST 2018: 收到客户端消息: 你好，你的微信公众号《闪电侠的博客》里的小闪对话系列博客太精彩了!
Wed Aug 15 08:08:20 CST 2018: 收到客户端消息: 感谢你用微信对话这种非常通俗的形式来讲技术，别断更啊!
```

10.6 总结

1. 定义了负责收发消息的Java对象中进行消息的收发。
2. 学习了Channel的attr()方法的实际用法：可以通过给Channel绑定属性来设置某些状态，获取某些状态，不需要额外的Map来维持。
3. 学习了如何在控制台获取消息并发送至服务端。
4. 实现了服务端回消息、客户端响应的逻辑。可以看到，这部分实际和前一章的登录流程有点类似。

10.7 思考

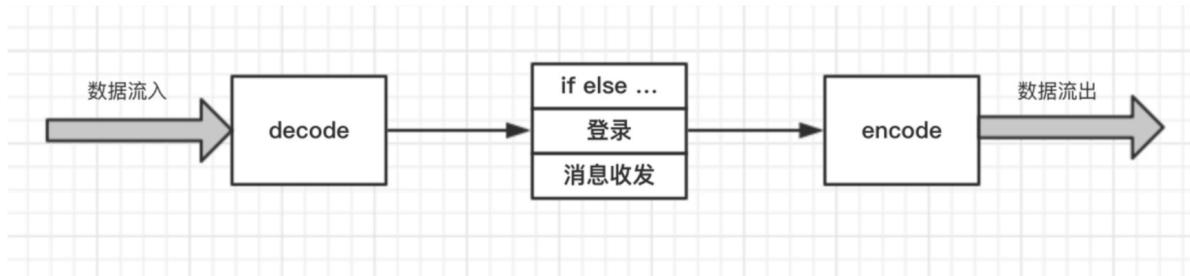
随着我们实现的指令越来越多，如何避免channelRead()中对指令处理的if else泛滥？

第11章

Pipeline与ChannelHandler

这一章，我们将学习Netty的两大核心组件：Pipeline与ChannelHandler。

我们在上一章的最后提出一个问题：如何避免if else泛滥？我们注意到，不管服务端还是客户端，处理流程大致都分为下图所示的几个步骤。



我们把这三类逻辑都写在一个类里，客户端写在ClientHandler，服务端写在ServerHandler，如果要做功能的扩展（比如我们要校验魔数，或者其他特殊逻辑），只能在一个类里修改，这个类就会变得越来越臃肿。

另外，每次发指令数据包都要手动调用编码器编码成ByteBuf，对于这类场景的编码优化，我们能想到的办法自然是模块化处理，不同的逻辑放置到单独的类中来处理，最后将这些逻辑串联起来，形成一个完整的逻辑处理链。

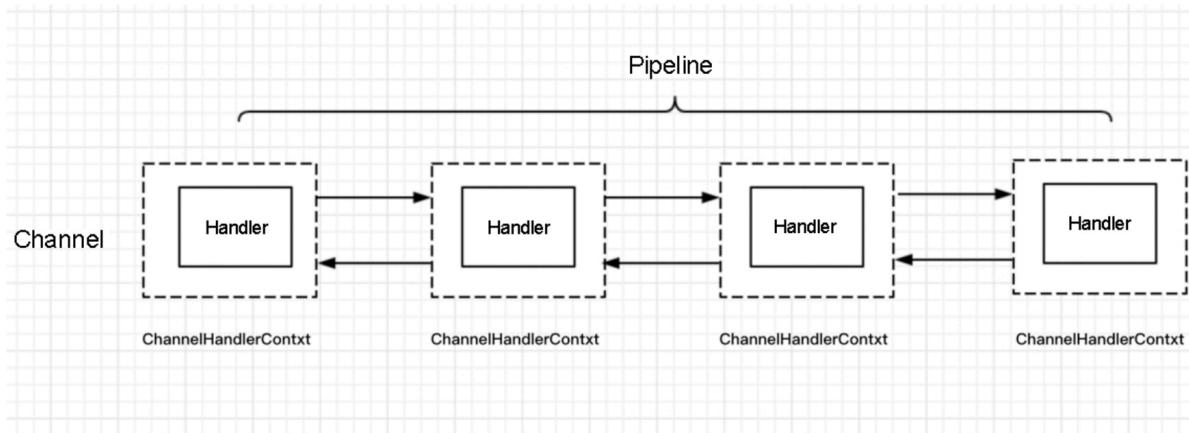
Netty中的Pipeline和ChannelHandler正是用来解决这个问题的。它通过责任链设计模式来组织代码逻辑，并且能够支持逻辑的动态添加和删除，Netty能够支持各类协议的扩展，比如HTTP、Websocket和Redis，靠的就是Pipeline和ChannelHandler。下面我们来学习这部分内容。

11.1 Pipeline与ChannelHandler的构成

无论从服务端来看，还是从客户端来看，在Netty的整个框架里面，一个连接对应着一个Channel。这个Channel的所有处理逻辑都在一个叫作ChannelPipeline的对象里，ChannelPipeline是双向链表结构，它和Channel之间是一对一的关系。

如下图所示，ChannelPipeline里的每个节点都是一个ChannelHandlerContext对象，这个对象能够获得和Channel相关的所有上下文信息。这个对象同时包含一个重要的

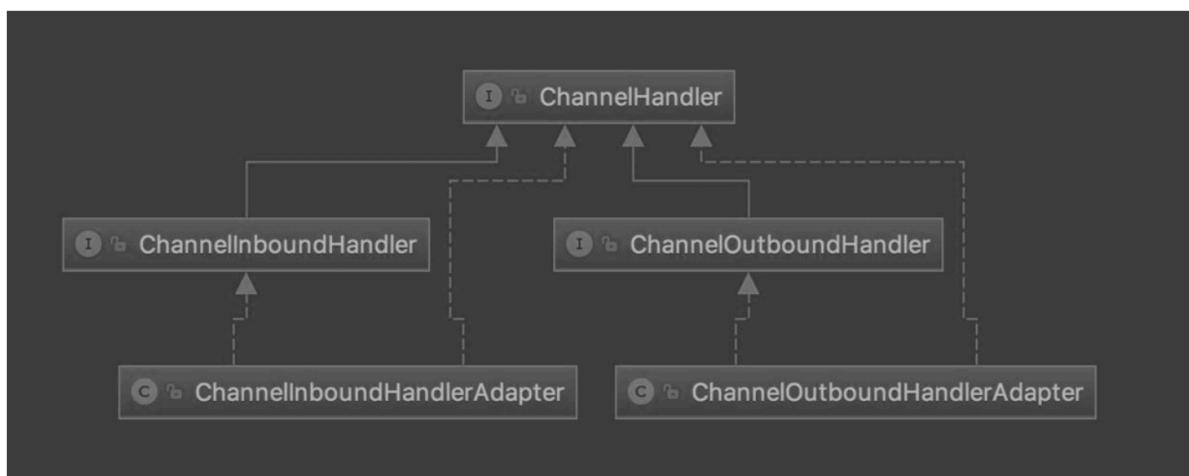
对象，那就是逻辑处理器ChannelHandler，每个ChannelHandler都处理一块独立的逻辑。



我们再来看ChannelHandler有哪些分类。

11.2 ChannelHandler的分类

由下图可以看到，ChannelHandler有两大子接口。



第一个子接口是ChannelInboundHandler，从字面意思可以猜到，它是处理读数据的逻辑。比如在一端读到一段数据，首先要解析这段数据，然后对这段数据做一系列逻辑处理，最终把响应写到对端。在组装响应之前的所有处理逻辑，都可以放置在一系列ChannelInboundHandler里处理，它的一个最重要的方法就是channelRead()。读者可以将ChannelInboundHandler的逻辑处理过程与TCP的七层协议解析过程联系起来，把收到的数据一层层地从物理层上升到应用层。

第二个子接口ChannelOutboundHandler是处理写数据的逻辑，它是定义一端在组装完响应之后把数据写到对端的逻辑。比如，我们封装好一个response对象后，有可能对这个response做一些其他特殊逻辑处理，然后编码成ByteBuf，最终写到对端。它最核心的方法就是write()，读者可以将ChannelOutboundHandler的逻辑处理过程与TCP的七层协议封装过程联系起来。我们在应用层组装响应之后，通过层层协议的封装，到底层的物理层。

这两个子接口分别有对应的默认实现：ChannelInboundHandlerAdapter和ChannelOutboundHandlerAdapter，它们分别实现了两大子接口的所有功能，在默认情况下会把读写事件传播到下一个Handler。

下面我们就用一个具体的Demo来学习这两大Handler的事件传播机制。

11.3 ChannelInboundHandler的事件传播

关于ChannelInboundHandler，我们用channelRead()作例子，来体验一下Inbound事件的传播。

我们在服务端的Pipeline添加3个ChannelInboundHandler。

NettyServer.java

```
serverBootstrap  
  
.childHandler(new ChannelInitializer<NioSocketChannel>() {  
  
protected void initChannel(NioSocketChannel ch) {  
  
ch.Pipeline().addLast(new InBoundHandlerA());  
  
ch.Pipeline().addLast(new InBoundHandlerB());  
  
ch.Pipeline().addLast(new InBoundHandlerC());  
  
}  
  
});
```

每个inboundHandler都继承自ChannelInboundHandlerAdapter，实现了channelRead()方法。

```
public class InBoundHandlerA extends ChannelInboundHandlerAdapter {

    @Override

    public void channelRead(ChannelHandlerContext ctx, Object msg)
throws Exception {

    System.out.println("InBoundHandlerA: " + msg);

    super.channelRead(ctx, msg);

}

}

public class InBoundHandlerB extends ChannelInboundHandlerAdapter {

    @Override

    public void channelRead(ChannelHandlerContext ctx, Object msg)
throws Exception {

    System.out.println("InBoundHandlerB: " + msg);

    super.channelRead(ctx, msg);

}

}

public class InBoundHandlerC extends ChannelInboundHandlerAdapter {

    @Override

    public void channelRead(ChannelHandlerContext ctx, Object msg)
throws Exception {

    System.out.println("InBoundHandlerC: " + msg);

}
```

```
super.channelRead(ctx, msg);

    }

}
```

在channelRead()方法里，我们打印当前Handler的信息，调用父类的channelRead()方法。而父类的channelRead()方法会自动调用下一个inboundHandler的channelRead()方法，并且会把当前inboundHandler里处理完毕的对象传递到下一个inboundHandler，上述例子中传递的对象都是同一个msg。

我们通过addLast()方法为Pipeline添加inboundHandler，当然，除了这个方法，还有其他方法，感兴趣的读者可以自行浏览Pipeline的API，这里我们添加的顺序为A ->B->C。最后来看一下控制台的输出，如下图所示。



```
Run: NettyServer NettyClient
/Library/Java/JavaVirtualMachines/jdk1.8.0_72.jdk/Contents/Home/bin/java ...
objc[29983]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines/jdk1.8.0_72.jdk/Contents/Home/bin/java ...
Thu Aug 16 06:43:26 CST 2018: 端口[8000]绑定成功!
InBoundHandlerA: PooledUnsafeDirectByteBuf(ridx: 0, widx: 96, cap: 1024)
InBoundHandlerB: PooledUnsafeDirectByteBuf(ridx: 0, widx: 96, cap: 1024)
InBoundHandlerC: PooledUnsafeDirectByteBuf(ridx: 0, widx: 96, cap: 1024)
```

由上图可以看到，inboundHandler的执行顺序与通过addLast()方法添加的顺序保持一致。我们再来看outboundHandler的事件传播。

11.4 ChannelOutboundHandler的事件传播

关于ChannelOutboundHandler，我们用write()作例子，来体验一下Outbound事件的传播。

我们继续在服务端的Pipeline添加3个ChannelOutboundHandler。

```
serverBootstrap

    .childHandler(new ChannelInitializer<NioSocketChannel>() {
        protected void initChannel(NioSocketChannel ch) {
            // inbound, 处理读数据的逻辑链
```

```

ch.Pipeline().addLast(new InboundHandlerA());

ch.Pipeline().addLast(new InboundHandlerB());

ch.Pipeline().addLast(new InboundHandlerC());

// outbound, 处理写数据的逻辑链

ch.Pipeline().addLast(new OutboundHandlerA());

ch.Pipeline().addLast(new OutboundHandlerB());

ch.Pipeline().addLast(new OutboundHandlerC());

}

}

);

```

每个outboundHandler都继承自ChannelOutboundHandlerAdapter，实现了write()方法。

```

public class OutboundHandlerA extends ChannelOutboundHandlerAdapter {

@Override

public void write(ChannelHandlerContext ctx, Object msg,
ChannelPromise promise)

throws Exception {

System.out.println("OutboundHandlerA: " + msg);

super.write(ctx, msg, promise);

}

}

public class OutboundHandlerB extends ChannelOutboundHandlerAdapter {

@Override

```

```
public void write(ChannelHandlerContext ctx, Object msg,
ChannelPromise promise)

throws Exception {

    System.out.println("OutboundHandlerB: " + msg);

    super.write(ctx, msg, promise);

}

}

public class OutboundHandlerC extends ChannelOutboundHandlerAdapter {

    public void write(ChannelHandlerContext ctx, Object msg,
ChannelPromise promise)

throws Exception {

    System.out.println("OutboundHandlerC: " + msg);

    super.write(ctx, msg, promise);

}

}
```

在write()方法里，我们打印当前Handler的信息，调用父类的write()方法。而父类的write()方法会自动调用下一个outboundHandler的write()方法，并且会把当前outboundHandler里处理完毕的对象传递到下一个outboundHandler。

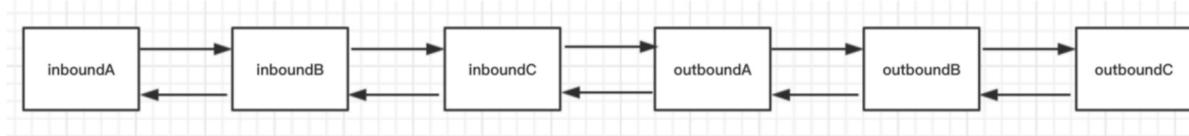
我们通过addLast()方法添加outboundHandler的顺序为A->B->C。最后来看控制台的输出，如下图所示。

```
objc[30141]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines/jdk1.8.0_72.jdk/Contents/Home/bin/java ...
Connected to the target VM, address: '127.0.0.1:63840', transport: 'socket'
Thu Aug 16 06:57:48 CST 2018: 端口[8000]绑定成功!
InBoundHandlerA: PooledUnsafeDirectByteBuf(ridx: 0, widx: 96, cap: 1024)
InBoundHandlerB: PooledUnsafeDirectByteBuf(ridx: 0, widx: 96, cap: 1024)
InBoundHandlerC: PooledUnsafeDirectByteBuf(ridx: 0, widx: 96, cap: 1024)
OutBoundHandlerC: PooledUnsafeDirectByteBuf(ridx: 0, widx: 96, cap: 1024)
OutBoundHandlerB: PooledUnsafeDirectByteBuf(ridx: 0, widx: 96, cap: 1024)
OutBoundHandlerA: PooledUnsafeDirectByteBuf(ridx: 0, widx: 96, cap: 1024)
```

由上图可以看到，outboundHandler的执行顺序与添加的顺序相反，这是为什么呢？这就要说到Pipeline的结构和执行顺序了。

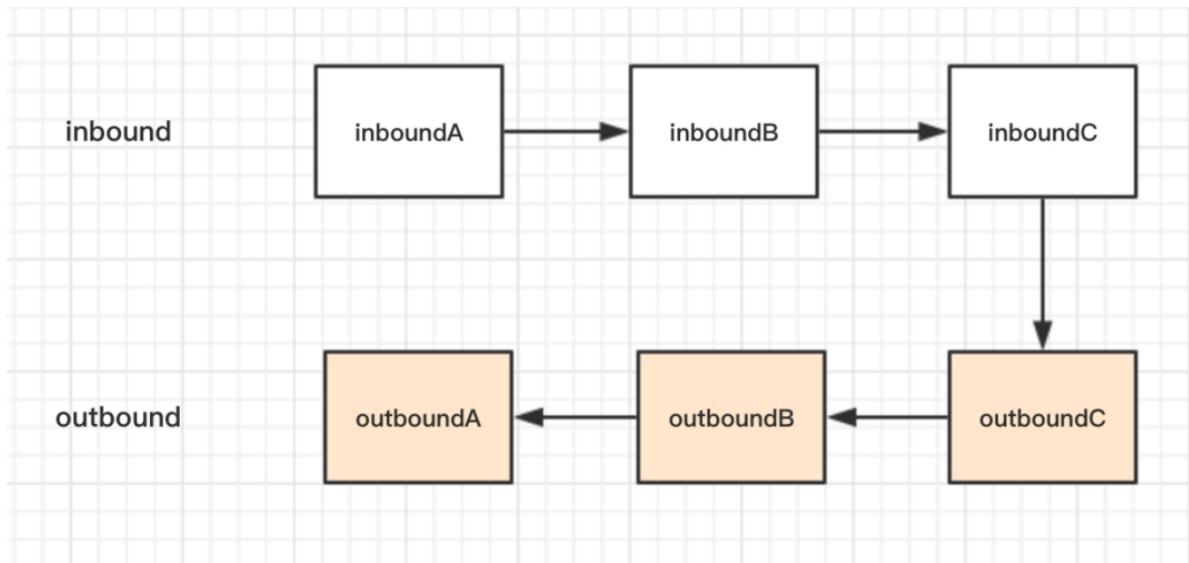
Pipeline的结构

如下图所示，不管我们定义的是哪种类型的Handler，最终它们都以双向链表的方式连接。实际链表的节点是ChannelHandlerContext，为了让结构清晰突出，可以直接把节点看作ChannelHandler。



Pipeline的执行顺序

虽然两种类型的Handler在一个双向链表里，但是这两类Handler的分工是不一样的，inboundHandler的事件通常只会传播到下一个inboundHandler，outboundHandler的事件通常只会传播到下一个outboundHandler，两者相互不受干扰，如下图所示。



关于Pipeline与ChannelHandler相关的事件传播就讲完了。下一章，我们会了解几种特殊的ChannelHandler，并且使用这几种特殊的ChannelHandler来改造客户端和服务端逻辑，解决if else泛滥的问题。

11.5 总结

- 1.通过前面编写客户端与服务端处理逻辑，引出了Pipeline和ChannelHandler的概念。
- 2.ChannelHandler分为Inbound和Outbound两种类型的接口，分别是处理数据读与数据写的逻辑，可与TCP协议栈处理数据的两个方向联系起来。
- 3.两种类型的Handler均有相应的默认实现，默认会把事件传递到下一个Handler，这里的传递事件其实就是把本Handler的处理结果传递到下一个Handler继续处理。
- 4.inboundHandler的执行顺序与实际的添加顺序相同，而outboundHandler则相反。

11.6 思考

- 1.参考本章的例子，如果我们往Pipeline里添加Handler的顺序不变，要在控制台打印出inboundA->inboundC->outboundB->outboundA，该如何实现？
- 2.如何在每个Handler里都打印上一个Handler处理结束的时间点？

第12章

构建客户端与服务端的Pipeline

通过上一章的学习，我们已经了解了Pipeline和ChannelHandler的基本概念。本章使用上一章的理论知识来重新构建客户端和服务端的Pipeline，把复杂的逻辑从单独的一个ChannelHandler中抽取出来。

Netty内置了很多开箱即用的ChannelHandler，我们通过学习Netty内置的ChannelHandler来逐步构建Pipeline。

12.1 ChannelInboundHandlerAdapter与 ChannelOutboundHandlerAdapter

首先是ChannelInboundHandlerAdapter，这个适配器主要用于实现其接口ChannelInboundHandler的所有方法，这样我们在编写自己的Handler时就不需要实现Handler里的每一种方法，而只需要实现我们所关心的方法即可。在默认情况下，对于ChannelInboundHandlerAdapter，我们比较关心的是它的如下方法。

ChannelInboundHandlerAdapter.java

```
@Override  
  
public void channelRead(ChannelHandlerContext ctx, Object msg) throws  
Exception {  
  
    ctx.fireChannelRead(msg);  
  
}
```

它的作用就是接收上一个Handler的输出，这里的msg就是上一个Handler的输出。大家也可以看到，默认情况下的Adapter会通过fireChannelRead()方法直接把上一个Handler的输出结果传递到下一个Handler。

与ChannelInboundHandlerAdapter类似的类是ChannelOutboundHandlerAdapter，它的核心方法如下。

ChannelOutboundHandlerAdapter.java

```
@Override

public void write(ChannelHandlerContext ctx, Object msg,
ChannelPromise promise) throws

Exception {

    ctx.write(msg, promise);

}
```

在默认情况下，这个Adapter也会把对象传递到下一个Outbound节点，它的传播顺序与inboundHandler相反，这里不再展开介绍。

我们往Pipeline添加的第一个Handler的channelRead()方法中，msg对象其实就是ByteBuf。服务端在接收到数据之后，应该要做的第一步逻辑就是把这个ByteBuf进行解码，然后把解码后的结果传递到下一个Handler，如下所示。

```
@Override

public void channelRead(ChannelHandlerContext ctx, Object msg) throws

Exception {

    ByteBuf requestByteBuf = (ByteBuf) msg;

    // 解码

    Packet packet = PacketCodeC.INSTANCE.decode(requestByteBuf);

    // 解码后的对象传递到下一个Handler

    ctx.fireChannelRead(packet)

}
```

在开始解码之前，我们先来了解一下另外一个特殊的Handler。

12.2 ByteToMessageDecoder

通常情况下，无论在客户端还是在服务端，当我们收到数据后，首先要做就是把二进制数据转换到Java对象，所以Netty很贴心地提供了一个父类，来专门做这个事

情。我们看一下如何使用这个类来实现服务端的解码。

```
public class PacketDecoder extends ByteToMessageDecoder {  
  
    @Override  
  
    protected void decode(ChannelHandlerContext ctx, ByteBuf in, List  
out) {  
  
        out.add(PacketCodeC.INSTANCE.decode(in));  
  
    }  
  
}
```

当继承了ByteToMessageDecoder这个类之后，只需要实现decode()方法即可。大家可以看到，这里的in传递进来的时候就已经是ByteBuf类型，所以不再需要强转。第三个参数是List类型，我们通过向这个List里面添加解码后的结果对象，就可以自动实现结果向下一个Handler传递，这样就实现了解码的逻辑Handler。

另外，值得注意的是，对于Netty里的ByteBuf，我们使用4.1.6.Final版本，默认情况下用的是堆外内存。在ByteBuf一章中提到，堆外内存需要我们自行释放，在解码的例子中，其实我们已经漏掉了这个操作，这一点是非常致命的。随着程序运行得越来越久，内存泄漏的问题就慢慢暴露出来了，而这里我们使用ByteToMessageDecoder，Netty会自动进行内存的释放，我们不用操心太多的内存管理方面的逻辑。

通过解码器把二进制数据转换到Java对象即指令数据包之后，就可以针对每一种指令数据包编写逻辑了。

12.3 SimpleChannelInboundHandler

回顾一下前面处理Java对象的逻辑。

```
if (packet instanceof LoginRequestPacket) {  
  
    // ...  
  
} else if (packet instanceof MessageRequestPacket) {  
  
    // ...
```

```
} else if ...
```

通过if else分支进行逻辑的处理，当要处理的指令越来越多的时候，代码会显得越来越臃肿，我们可以通过给Pipeline添加多个Handler（ChannelInboundHandlerAdapter的子类）来解决过多的if else问题。

XXXHandler.java

```
if (packet instanceof XXXPacket) {  
  
    // ...处理  
  
} else {  
  
    ctx.fireChannelRead(packet);  
  
}
```

这样的一个好处就是，每次添加一个指令处理器，其逻辑处理的框架都是一致的。

但是，大家应该也注意到了，我们编写指令处理Handler的时候，依然编写了一段其实可以不用关心的if else判断，然后手动传递无法处理的对象（XXXPacket）至下一个指令处理器，这也是一段重复度极高的代码。因此，基于这种考虑，Netty抽象出了一个SimpleChannelInboundHandler对象，自动实现了类型判断和对象传递，这样我们的应用代码就可以专注于业务逻辑。

下面来看如何使用SimpleChannelInboundHandler简化指令处理逻辑。

LoginRequestHandler.java

```
public class LoginRequestHandler extends  
SimpleChannelInboundHandler<LoginRequestPacket>  
  
{  
  
    @Override  
  
    protected void channelRead0(ChannelHandlerContext ctx,  
    LoginRequestPacket  
  
    loginRequestPacket) {
```

```
// 登录逻辑  
}  
}  
}
```

从字面意思可以看到，`SimpleChannelInboundHandler`的使用非常简单。我们在继承这个类的时候，给它传递一个泛型参数，然后在`channelRead0()`方法里，不用再通过if逻辑来判断当前对象是否是本Handler可以处理的对象，也不用强转，不用往下传递本Handler处理不了的对象，这一切都已经交给父类`SimpleChannelInboundHandler`来实现，我们只需要专注于我们要处理的业务逻辑即可。

上面的`LoginRequestHandler`是用来处理登录的逻辑，同理，我们可以很轻松地编写一个消息处理逻辑处理器。

MessageRequestHandler.java

```
public class MessageRequestHandler extends  
SimpleChannelInboundHandler<MessageRequestPacket> {  
  
    @Override  
  
    protected void channelRead0(ChannelHandlerContext ctx,  
MessageRequestPacket  
  
messageRequestPacket) {  
  
    }  
}
```

12.4 MessageToByteEncoder

在前面的章节中，我们已经实现了登录和消息处理逻辑。处理完登录和消息这两类指令之后，我们都会给客户端返回一个响应。在写响应之前，需要把响应对象编码成`ByteBuf`，结合本节内容，最后的逻辑框架如下。

```
public class LoginRequestHandler extends  
SimpleChannelInboundHandler<LoginRequestPacket  
  
{
```

```
    @Override

    protected void channelRead0(ChannelHandlerContext ctx,
LoginRequestPacket

loginRequestPacket)  {

    LoginResponsePacket loginResponsePacket = login(loginRequestPacket);

    ByteBuf responseByteBuf = PacketCodeC.INSTANCE.encode(ctx.alloc(),
loginResponsePacket);

    ctx.channel().writeAndFlush(responseByteBuf);

}

}

public class MessageRequestHandler extends

SimpleChannelInboundHandler<MessageRequestPacket>  {

    @Override

    protected void channelRead0(ChannelHandlerContext ctx,
MessageRequestPacket

messageRequestPacket)  {

    MessageResponsePacket messageResponsePacket = receiveMessage

(messageRequestPacket);

    ByteBuf responseByteBuf = PacketCodeC.INSTANCE.encode(ctx.alloc(),
messageRequestPacket);
```

```
    ctx.channel().writeAndFlush(responseByteBuf);

    }

}
```

读者应该注意到了，在上述代码中，处理每一种指令完成之后的逻辑都是类似的，都需要先进行编码，然后调用writeAndFlush()将数据写到客户端。这个编码的过程其实也是重复的逻辑，而且在编码过程中，我们还需要手动创建一个ByteBuf，过程如下。

PacketCodeC.java

```
public ByteBuf encode(ByteBufAllocator byteBufAllocator, Packet
packet) {

    // 1. 创建 ByteBuf 对象

    ByteBuf byteBuf = byteBufAllocator.ioBuffer();

    // 2. 序列化 Java 对象

    // 3. 实际编码过程

    return byteBuf;

}
```

Netty提供了一个特殊的ChannelHandler来专门处理编码逻辑，不需要每一次将响应写到对端的时候都调用一次编码逻辑进行编码，也不需要自行创建ByteBuf。这个类被叫作MessageToByteEncoder，从字面意思可以看出，它的功能就是将对象转换到二进制数据。

使用MessageToByteEncoder来实现编码逻辑的过程如下。

```
public class PacketEncoder extends MessageToByteEncoder<Packet> {

    @Override

    protected void encode(ChannelHandlerContext ctx, Packet packet,
ByteBuf out) {
```

```
    PacketCodeC.INSTANCE.encode(out, packet);

}

}
```

PacketEncoder继承自MessageToByteEncoder，泛型参数Packet表示这个类的作用是实现Packet类型对象到二进制数据的转换。

这里我们只需要实现encode()方法。在这个方法里，第二个参数是Java对象，而第三个参数是ByteBuf对象，我们要做的事情就是，把Java对象的字段写到ByteBuf对象，而不再需要自行去分配ByteBuf对象。因此，大家注意到，PacketCodeC的encode()方法的定义也改了，下面是更改前后的对比。

PacketCodeC.java

```
// 更改前的定义

public ByteBuf encode(ByteBufAllocator byteBufAllocator, Packet
packet) {

    // 1. 创建 ByteBuf 对象

    ByteBuf byteBuf = byteBufAllocator.ioBuffer();

    // 2. 序列化 Java 对象

    // 3. 实际编码过程

    return byteBuf;

}

// 更改后的定义

public void encode(ByteBuf byteBuf, Packet packet) {

    // 1. 序列化 Java 对象
```

```
// 2. 实际编码过程
```

```
}
```

PacketCodeC不再需要手动创建ByteBuf对象，不再需要把创建完的ByteBuf对象进行返回。当我们向Pipeline中添加这个编码器后，在指令处理完毕之后就只需要调用writeAndFlush()把Java对象写出去即可。

```
public class LoginRequestHandler extends SimpleChannelInboundHandler<LoginRequestPacket> {

    @Override
    protected void channelRead0(ChannelHandlerContext ctx, LoginRequestPacket loginRequestPacket) {
        ctx.channel().writeAndFlush(login(loginRequestPacket));
    }
}

public class MessageRequestHandler extends SimpleChannelInboundHandler<MessageResponsePacket> {
    @Override
    protected void channelRead0(ChannelHandlerContext ctx, MessageResponsePacket messageRequestPacket) {

```

```

    ctx.channel().writeAndFlush(receiveMessage(messageRequestPacket));

}

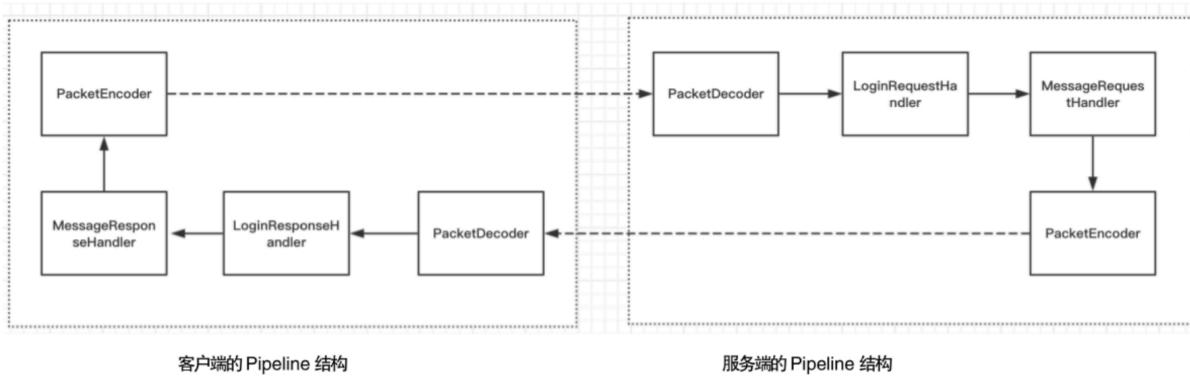
}

```

通过前面的分析，可以看到，Netty为了让我们的逻辑更为清晰简洁，做了很多工作，能直接用Netty自带的Handler来解决的问题，不再需要重复造轮子。在接下来的章节中，我们会继续探讨Netty还有哪些开箱即用的Handler。

12.5 构建客户端与服务端的Pipeline

分析完服务端的Pipeline的Handler组成结构，相信读者也不难自行分析出客户端的Pipeline的Handler结构。最后我们来看一下客户端和服务端完整的Pipeline的Handler结构，如下图所示。



对应代码如下。

客户端

bootstrap

```

.handler(new ChannelInitializer<SocketChannel>() {
    @Override
    public void initChannel(SocketChannel ch) {
        ch.Pipeline().addLast(new PacketDecoder());
    }
})

```

```
        ch.Pipeline().addLast(new LoginResponseHandler());  
  
        ch.Pipeline().addLast(new MessageResponseHandler());  
  
        ch.Pipeline().addLast(new PacketEncoder());  
  
    }  
}  
};
```

服务端

```
serverBootstrap  
  
.childHandler(new ChannelInitializer<NioSocketChannel>() {  
  
protected void initChannel(NioSocketChannel ch) {  
  
    ch.Pipeline().addLast(new PacketDecoder());  
  
    ch.Pipeline().addLast(new LoginRequestHandler());  
  
    ch.Pipeline().addLast(new MessageRequestHandler());  
  
    ch.Pipeline().addLast(new PacketEncoder());  
  
}  
}  
};
```

12.6 总结

本章我们学习了用Netty内置的ChannelHandler来逐步构建服务端的Pipeline，通过内置的ChannelHandler可以减少很多重复的逻辑。

1. 基于ByteToMessageDecoder，可以实现自定义解码，而不用关心ByteBuf的强转和解码结果的传递。
2. 基于SimpleChannelInboundHandler，可以实现每一种指令的处理，不再需要强转，不再有冗长乏味的if else逻辑，不再需要手动传递对象。

3. 基于MessageToByteEncoder，可以实现自定义编码，不用关心ByteBuf的创建，不用每次向对端写Java对象都进行一次编码。

12.7 思考

在LoginRequestHandler和MessageRequestHandler的channelRead0()方法中，第二个参数对象 (XXXRequestPacket) 是从哪里传递过来的？

第13章

拆包/粘包理论与解决方案

本章我们来学习一下Netty里拆包和粘包的概念，并且学习如何选择适合我们的应用程序的拆包器。

13.1 拆包/粘包例子

我们先来看一个例子，选择客户端与服务端双向通信这一章节的代码，然后做适当修改。

客户端FirstClientHandler

```
public class FirstClientHandler extends ChannelInboundHandlerAdapter {  
  
    @Override public void channelActive(ChannelHandlerContext ctx) {  
  
        for (int i = 0; i < 1000; i++) {  
  
            ByteBuf buffer = getByteBuf(ctx);  
  
            ctx.channel().writeAndFlush(buffer);  
  
        }  
    }  
  
    private ByteBuf getByteBuf(ChannelHandlerContext ctx) {  
  
        byte [] bytes = "你好，欢迎关注我的微信公众号，《闪电侠的博客》！ ".getBytes  
(Charset.forName("utf-8"));  
  
        ByteBuf buffer = ctx.alloc().buffer();  
  
        buffer.writeBytes(bytes);  
    }  
}
```

```
    return buffer;

}

}
```

客户端在连接建立成功之后，使用一个for循环，不断地向服务端写数据。

服务端FirstServerHandler

```
public class FirstServerHandler extends ChannelInboundHandlerAdapter {

    @Override public void channelRead(ChannelHandlerContext ctx, Object msg) {

        ByteBuf byteBuf = (ByteBuf) msg;

        System.out.println(new Date() + ": 服务端读到数据 -> " +
byteBuf.toString (Charset.forName("utf-8")));

    }

}
```

服务端在收到数据之后，仅仅把数据打印出来。读者可以花几分钟时间思考一下，服务端的输出会是什么样子的？

可能很多读者觉得服务端会输出1000次“你好，欢迎关注我的微信公众号，《闪电侠的博客》！”，然而实际上服务端却是这样输出的，如下图所示。

从服务端的控制台输出可以看出，存在3种类型的输出。

- 1.一种是正常的字符串输出。
 - 2.一种是多个字符串“粘”在了一起，我们定义这种ByteBuf为粘包。
 - 3.一种是一个字符串被“拆”开，形成一个破碎的包，我们定义这种ByteBuf为半包。

13.2 为什么会有粘包、半包现象

尽管我们在应用层面使用了Netty，但是操作系统只认TCP协议；尽管我们的应用层按照ByteBuf为单位来发送数据，但是到了底层操作系统，仍然是按照字节流发送数据的，因此，数据到了服务端，也按照字节流的方式读入，然后到了Netty应用层面，重新拼装成ByteBuf。这里的ByteBuf与客户端按照顺序发送的ByteBuf可能是不对等的。因此，我们需要在客户端根据自定义协议来组装应用层的数据包，然后在服务端根据应用层的协议来组装数据包，这个过程通常在服务端被称为拆包，而在客户端被称为粘包。

拆包和粘包是相对的，一端粘了包，另外一端就需要将粘过的包拆开。举个例子，发送端将三个数据包粘成两个TCP数据包发送到接收端，接收端就需要根据应用协议将两个数据包重新拆分成三个数据包。

13.3 拆包的原理

在没有Netty的情况下，用户如果自己需要拆包，基本原理就是不断地从TCP缓冲区中读取数据，每次读取完都需要判断是否是一个完整的数据包。

1.如果当前读取的数据不足以拼接成一个完整的业务数据包，那就保留该数据，继续从TCP缓冲区中读取，直到得到一个完整的数据包。

2.如果当前读到的数据加上已经读取的数据足够拼接成一个数据包，那就将已经读取的数据拼接上本次读取的数据，构成一个完整的业务数据包传递到业务逻辑，多余的数据仍然保留，以便和下次读到的数据尝试拼接。

如果我们自己实现拆包，那么这个过程将会非常麻烦。每一种自定义协议都需要自己实现，还需要考虑各种异常，而Netty自带的一些开箱即用的拆包器已经完全满足我们的需求了。下面介绍Netty有哪些自带的拆包器。

13.4 Netty自带的拆包器

13.4.1 固定长度的拆包器**FixedLengthFrameDecoder**

如果应用层协议非常简单，每个数据包的长度都是固定的，比如100，那么只需要把这个拆包器加到Pipeline中，Netty就会把一个个长度为100的数据包（ByteBuf）传递到下一个ChannelHandler。

13.4.2 行拆包器**LineBasedFrameDecoder**

从字面意思来看，发送端发送数据包的时候，每个数据包之间以换行符作为分隔，接收端通过LineBasedFrameDecoder将粘过的ByteBuf拆分成一个个完整的应用层数据包。

13.4.3 分隔符拆包器**DelimiterBasedFrameDecoder**

DelimiterBasedFrameDecoder是行拆包器的通用版本，只不过我们可以自定义分隔符。

13.4.4 基于长度域的拆包器**LengthFieldBasedFrameDecoder**

最后一种拆包器是最通用的一种拆包器，只要你的自定义协议中包含长度域字段，均可以使用这个拆包器来实现应用层拆包。由于上面3种拆包器比较简单，读者可以自行写出Demo。接下来，我们就结合自定义协议，来学习如何使用基于长度域的拆包器来拆解数据包。

13.5 如何使用LengthFieldBasedFrameDecoder

首先，我们来回顾一下自定义协议。

| | | | | | |
|-----------------|--------|-------|-----|------|--------|
| 魔数 (0x12345678) | 版本号(1) | 序列化算法 | 指令 | 数据长度 | 数据 |
| 4字节 | 1字节 | 1字节 | 1字节 | 4字节 | N 字节 |

详细的协议分析参考客户端与服务端通信协议编解码一节，这里不再赘述。

关于拆包，我们只需要关注以下3点。

- 1.在我们的自定义协议中，长度域在整个数据包的哪个地方。用专业术语来说，就是长度域相对整个数据包的偏移量是多少，这里显然是 $4+1+1+1=7$ 。
- 2.另外需要关注的就是，长度域的长度是多少，这里显然是4。
- 3.有了长度域偏移量和长度域的长度，我们就可以构造一个拆包器。

```
new LengthFieldBasedFrameDecoder(Integer.MAX_VALUE, 7, 4);
```

其中，第一个参数指的是数据包的最大长度，第二个参数指的是长度域的偏移量，第三个参数指的是长度域的长度。写好这样一个拆包器之后，只需要在Pipeline的最前面加上这个拆包器即可。

由于这类拆包器使用最为广泛，想深入学习的读者可以参考笔者在简书上的文章。

下面我们重新组织一下服务端和客户端的Pipeline。

服务端

```
ch.pipeline().addLast(new  
LengthFieldBasedFrameDecoder(Integer.MAX_VALUE, 7, 4));  
  
ch.pipeline().addLast(new PacketDecoder());  
  
ch.pipeline().addLast(new LoginRequestHandler());  
  
ch.pipeline().addLast(new MessageRequestHandler());
```

```
ch.pipeline().addLast(new PacketEncoder());
```

客户端

```
ch.pipeline().addLast(new LengthFieldBasedFrameDecoder(Integer.MAX_VALUE, 7, 4));  
  
ch.pipeline().addLast(new PacketDecoder());  
  
ch.pipeline().addLast(new LoginResponseHandler());  
  
ch.pipeline().addLast(new MessageResponseHandler());  
  
ch.pipeline().addLast(new PacketEncoder());
```

这样，在后续PacketDecoder进行decode操作的时候，ByteBuf就是一个个完整的自定义协议数据包了。

LengthFieldBasedFrameDecoder有很多重载的构造参数，由于篇幅原因，这里不再展开介绍，关于LengthFieldBasedFrameDecoder的详细使用可参考**笔者在简书上的文章**，对原理感兴趣的读者可以参考**笔者在慕课网的视频**，了解了详细的使用方法之后，就可以有针对性地根据自己的自定义协议来构造LengthFieldBasedFrameDecoder。

13.6 拒绝非本协议连接

不知道大家还记得，我们在设计协议的时候为什么在数据包的开头加上一个魔数。我们设计魔数的原因是尽早屏蔽非本协议的客户端，通常在第一个Handler处理这段逻辑。接下来的做法是每个客户端发过来的数据包都做一次快速判断，判断当前发来的数据包是否满足我们的自定义协议。

我们只需要继承自LengthFieldBasedFrameDecoder的decode()方法，然后在decode之前判断前4字节是否等于我们定义的魔数0x12345678即可。

```
public class Spliter extends LengthFieldBasedFrameDecoder {  
  
    private static final int LENGTH_FIELD_OFFSET = 7;  
  
    private static final int LENGTH_FIELD_LENGTH = 4;  
  
    public Spliter() {
```

```
super(Integer.MAX_VALUE, LENGTH_FIELD_OFFSET, LENGTH_FIELD_LENGTH);

}

@Override protected Object decode(ChannelHandlerContext ctx, ByteBuf
in) throws Exception {

// 屏蔽非本协议的客户端

if (in.getInt(in.readerIndex()) != PacketCodeC.MAGIC_NUMBER) {

ctx.channel().close();

return null;

}

return super.decode(ctx, in);

}

}
```

为什么可以在decode()方法中写这段逻辑？是因为在decode()方法中，每次第二个参数in传递进来的时候，均是一个数据包的开头。

我们只需要替换如下代码即可。

```
//ch.pipeline().addLast(new
LengthFieldBasedFrameDecoder(Integer.MAX_VALUE, 7, 4));

// 替换为

ch.pipeline().addLast(new Spliter());
```

我们再来实验一下，如下图所示。

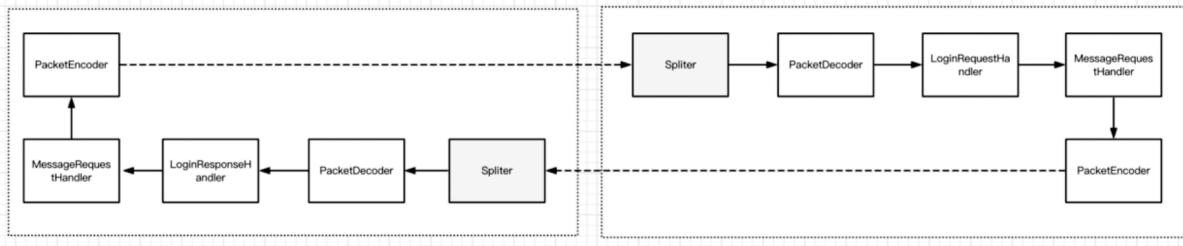
```
~ telnet 127.0.0.1 8000 ← 连上服务端
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.

^]
telnet> send ?
ao          Send Telnet Abort output
ayt         Send Telnet 'Are You There' ←
brk          Send Telnet Break
ec           Send Telnet Erase Character
el           Send Telnet Erase Line
escape       Send current escape character
ga           Send Telnet 'Go Ahead' sequence
ip           Send Telnet Interrupt Process
nop          Send Telnet 'No operation'
eor          Send Telnet 'End of Record'
abort        Send Telnet 'Abort Process'
susp         Send Telnet 'Suspend Process'
eof          Send Telnet End of File Character
synch        Perform Telnet 'Synch operation'
getstatus    Send request for STATUS
?            Display send options
telnet> send ayt   发送字符串 "Are you There"
Connection closed by foreign host. ← 被关闭了
~ |
```

由上图可以看到，使用telnet连接上服务端之后（与服务端建立了连接），向服务端发送一段字符串，由于这段字符串不符合我们的自定义协议，于是在第一时间，服务端就关闭了这个连接。

13.7 客户端和服务端的Pipeline结构

至此，客户端和服务端的Pipeline结构如下图所示。



客户端的Pipeline结构 服务端的Pipeline结构

13.8 总结

1. 我们通过一个例子来理解为什么要有拆包器。其实拆包器的作用就是，根据我们的自定义协议，把数据拼装成一个个符合自定义数据包大小的ByteBuf，然后发送到自定义协议的解码器中去解码。
2. Netty自带的拆包器包括基于固定长度的拆包器、基于换行符和自定义分隔符的拆包器，还有最重要的一种是基于长度域的拆包器。通常Netty自带的拆包器已完全满足我们的需求，无须重复造轮子。
3. 基于Netty自带的拆包器，我们可以在拆包之前判断当前连上的客户端是否支持自定义协议。如果不支持，可尽早关闭，节省资源。

13.9 思考

在IM完整的Pipeline中，如果我们不添加拆包器，客户端连续向服务端发送数据，会有什么现象发生？为什么会发生这种现象？

第14章

ChannelHandler的生命周期

在前面的章节中，对于ChannelHandler，我们的重点落在了读取数据相关的逻辑。本章，我们来学习ChannelHandler的其他方法，这些方法的执行是有顺序的，而这个执行顺序可以被称为ChannelHandler的生命周期。

14.1 ChannelHandler的生命周期详解

基于前面的代码，我们添加一个自定义ChannelHandler来测试一下各个回调方法的执行顺序。

对于服务端应用程序来说，我们这里讨论的ChannelHandler更多的是
ChannelInboundHandler，我们基于ChannelInboundHandlerAdapter，自定义了一个
Handler：LifeCyCleTestHandler。

LifeCyCleTestHandler.java

```
public class LifeCyCleTestHandler extends ChannelInboundHandlerAdapter
{
    @Override
    public void handlerAdded(ChannelHandlerContext ctx) throws Exception
    {
        System.out.println("逻辑处理器被添加: handlerAdded()");
        super.handlerAdded(ctx);
    }
    @Override
    public void channelRegistered(ChannelHandlerContext ctx) throws
Exception {
}
```

```
    System.out.println("channel 绑定到线程(NioEventLoop):  
channelRegistered());  
  
    super.channelRegistered(ctx);  
  
}  
  
@Override  
  
public void channelActive(ChannelHandlerContext ctx) throws Exception  
{  
  
    System.out.println("channel 准备就绪: channelActive()");  
  
    super.channelActive(ctx);  
  
}  
  
@Override  
  
public void channelRead(ChannelHandlerContext ctx, Object msg) throws  
Exception {  
  
    System.out.println("channel 有数据可读: channelRead()");  
  
    super.channelRead(ctx, msg);  
  
}  
  
@Override  
  
public void channelReadComplete(ChannelHandlerContext ctx) throws  
Exception {  
  
    System.out.println("channel 某次数据读完: channelReadComplete()");  
  
    super.channelReadComplete(ctx);  
  
}  
  
@Override
```

```
public void channelInactive(ChannelHandlerContext ctx) throws
Exception {
    System.out.println("channel 被关闭: channelInactive()");
    super.channelInactive(ctx);
}

@Override

public void channelUnregistered(ChannelHandlerContext ctx) throws
Exception {
    System.out.println("channel取消线程(NioEventLoop) 的绑定:
channelUnregistered()");
    super.channelUnregistered(ctx);
}

@Override

public void handlerRemoved(ChannelHandlerContext ctx) throws Exception
{
    System.out.println("逻辑处理器被移除: handlerRemoved()");
    super.handlerRemoved(ctx);
}

}
```

从上面的代码可以看到，我们在每个方法被调用的时候都会打印一段文字，然后把这个事件继续往下传播。最后把这个Handler添加到我们上章构建的Pipeline中。

```
// 前面代码省略

.childHandler(new ChannelInitializer<NioSocketChannel>() {

```

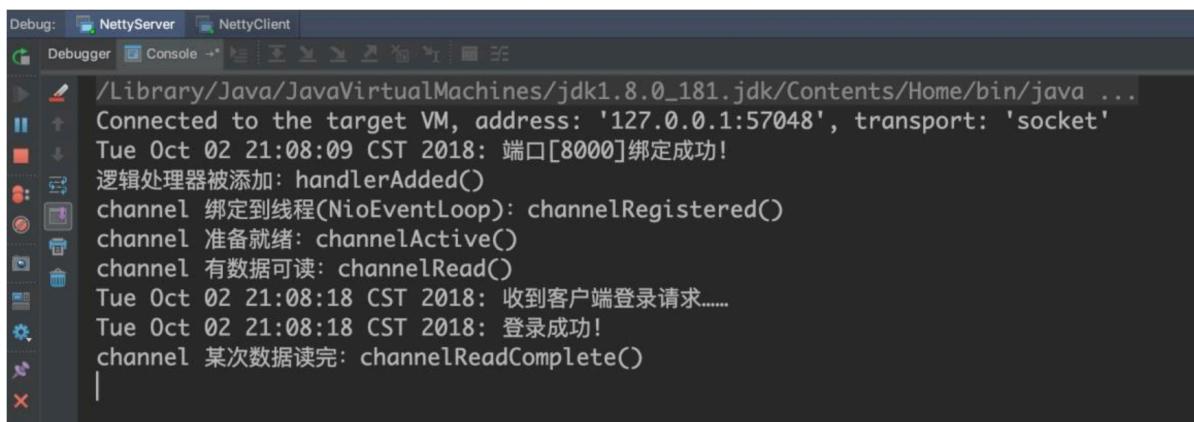
```

protected void initChannel(NioSocketChannel ch) {
    // 添加到第一个
    ch.pipeline().addLast(new LifeCycleTestHandler());
    ch.pipeline().addLast(new PacketDecoder());
    ch.pipeline().addLast(new LoginRequestHandler());
    ch.pipeline().addLast(new MessageRequestHandler());
    ch.pipeline().addLast(new PacketEncoder());
}

}

```

我们先运行NettyServer.java，然后运行NettyClient.java。这个时候，服务端控制台的输出如下图所示。



由上图可以看到，ChannelHandler回调方法的执行顺序为：

```

handlerAdded() -> channelRegistered() -> channelActive() ->
channelRead() ->

channelReadComplete()

```

下面我们来逐个解释每个回调方法的含义。

1.handlerAdded(): 指当检测到新连接之后，调用ch.pipeline().addLast(new LifeCycleTestHandler());之后的回调，表示在当前Channel中，已经成功添加了一个Handler处理器。

2.channelRegistered(): 这个回调方法表示当前Channel的所有逻辑处理已经和某个NIO线程建立了绑定关系，接收新的连接，然后创建一个线程来处理这个连接的读写，只不过在Netty里使用了线程池的方式，只需要从线程池里去抓一个线程绑定在这个Channel上即可。这里的NIO线程通常指NioEventLoop。

3.channelActive(): 当Channel的所有业务逻辑链准备完毕（即Channel的Pipeline中已经添加完所有的Handler），以及绑定好一个NIO线程之后，这个连接才真正被激活，接下来就会回调到此方法。

4.channelRead(): 客户端向服务端发送数据，每次都会回调此方法，表示有数据可读。

5.channelReadComplete(): 服务端每读完一次完整的数据，都回调该方法，表示数据读取完毕。

我们再把客户端关闭，这个时候对于服务端来说，其实就是Channel被关闭，如下图所示。

```
channel 被关闭: channelInactive()  
channel 取消线程(NioEventLoop) 的绑定: channelUnregistered()  
逻辑处理器被移除: handlerRemoved()
```

ChannelHandler回调方法的执行顺序为：

```
channelInactive() -> channelUnregistered() -> handlerRemoved()
```

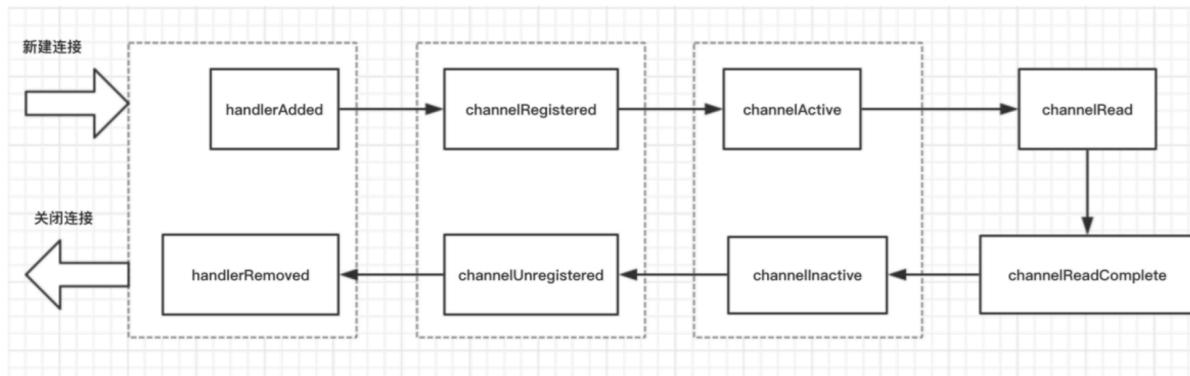
到这里，相信大家应该已经能够看到，这里回调方法的执行顺序是新连接建立时候的逆操作。下面我们来解释一下每个方法的含义。

1.channelInactive(): 表面上这个连接已经被关闭了，这个连接在TCP层面已经不再是ESTABLISH状态了。

2.channelUnregistered(): 既然连接已经被关闭，那么与这个连接绑定的线程就不需要对这个连接负责了。这个回调表明与这个连接对应的NIO线程移除了对这个连接的处理。

3.handlerRemoved(): 我们给这个连接添加的所有业务逻辑处理器都被移除。

最后，我们用下图来标识Channelhandler的生命周期。



光了解这些生命周期的回调方法其实是比较枯燥乏味的，接下来我们就看一下这些回调方法的使用场景。

14.2 ChannelHandler生命周期各回调方法的用法举例

Netty对于一个连接在各个不同状态下回调方法的定义还是比较细致的，好处就在于我们能够基于这个机制写出扩展性较好的应用程序。

14.2.1 ChannelInitializer的实现原理

仔细翻看一下服务端的启动代码，我们在给新连接定义Handler的时候，其实只是通过childHandler()方法给新连接设置了一个Handler。这个Handler就是ChannelInitializer，而在ChannelInitializer的initChannel()方法里，我们通过获得Channel对应的Pipeline，调用addLast()方法添加Handler。

NettyServer.java

```
.childHandler(new ChannelInitializer<NioSocketChannel>() {
    protected void initChannel(NioSocketChannel ch) {
        ch.pipeline().addLast(new LifeCycleTestHandler());
        ch.pipeline().addLast(new PacketDecoder());
        ch.pipeline().addLast(new LoginRequestHandler());
    }
});
```

```
        ch.pipeline().addLast(new MessageRequestHandler());  
  
        ch.pipeline().addLast(new PacketEncoder());  
  
    }  
};
```

这里的ChannelInitializer其实就利用了Netty的Handler生命周期中channelRegistered()与handlerAdded()两个特性，我们简单看下ChannelInitializer类的源码。

ChannelInitializer.java

```
protected abstract void initChannel(C ch) throws Exception;  
  
public final void channelRegistered(ChannelHandlerContext ctx)  
throws Exception {  
  
    // ...  
  
    initChannel(ctx);  
  
    // ...  
  
}  
  
public void handlerAdded(ChannelHandlerContext ctx) throws Exception  
{  
  
    // ...  
  
    if (ctx.channel().isRegistered()) {  
  
        initChannel(ctx);  
  
    }  
  
    // ...  
  
}
```

```
private boolean initChannel(ChannelHandlerContext ctx) throws
Exception {
    if (initMap.putIfAbsent(ctx, Boolean.TRUE) == null) {
        initChannel((C) ctx.channel());
        // ...
        return true;
    }
    return false;
}
```

这里，我们把非重点代码略去，逻辑会更加清晰一些。

1.ChannelInitializer定义了一个抽象的initChannel()方法，这个抽象方法由我们自行实现。我们在服务端启动流程里的实现逻辑就是往Pipeline里组织我们的Handler链。

2.handlerAdded()方法和channelRegistered()方法都会尝试调用initChannel()方法，initChannel()方法使用putIfAbsent()方法来防止initChannel()方法被调用多次。

3.如果读者Debug了ChannelInitializer的上述两个方法，就会发现，在handlerAdded()方法被调用的时候，Channel其实已经和某个线程绑定，所以就我们的应用程序来说，这里的channelRegistered()方法其实是多余的，那么为什么还要尝试调用一次呢？应该是担心我们自己写了一个类继承自ChannelInitializer，然后覆盖掉了handlerAdded()方法。这样即使覆盖掉，在channelRegistered()方法里还有机会再调一次initChannel()方法，把自定义的Handler都添加到Pipeline中去。

14.2.2 handlerAdded()方法与handlerRemoved()方法

这两个方法通常可以用于一些资源的申请和释放。

14.2.3 channelActive()方法与channelInActive()方法

1.对应用程序来说，这两个方法的含义是TCP连接的建立与释放。通常我们在这两个回调里统计单机的连接数，channelActive()方法被调用，连接数加一；channelInActive()方法被调用，连接数减一。

2.我们也可以在channelActive()方法中，实现对客户端连接IP黑白名单的过滤，具体就不展开介绍了。

14.2.4 channelRead()方法

我们在前面讲到拆包/粘包原理，服务端根据自定义协议来进行拆包，其实就是在这些方法里，每次读到一定数据，都会累加到一个容器里，然后判断是否能够拆出来一个完整的数据包。如果够就拆了之后往下进行传递。详细原理这里不过多展开，感兴趣的读者可以阅读本书第24章。

14.2.5 channelReadComplete()方法

在前面章节中，每次向客户端写数据的时候，都通过writeAndFlush()方法写数据并刷新到底层，其实这种方式并不是特别高效。我们可以把调用writeAndFlush()方法的地方都调用write()方法，然后在这个方法里调用ctx.channel().flush()方法，相当于批量刷新的机制。当然，如果你对性能要求没那么高，使用writeAndFlush()方法足矣。

14.3 总结

1.本章详细剖析了ChannelHandler（主要是ChannelInboundHandler）的各个回调方法、连接的建立和关闭，执行回调方法有一个逆向过程。

2.每一种回调方法都有其各自的用法，但是有的时候某些回调方法的使用边界有些模糊，恰当地使用回调方法来处理不同的逻辑，可以使你的应用程序更为简洁。

14.4 思考

1.如何在服务端每隔一秒输出当前客户端的连接数？

2.统计客户端的入口流量，以字节为单位。

第15章

使用ChannelHandler的热插拔实现客户端身份校验

在前面的章节中，细心的读者可能会注意到，客户端连上服务端之后，即使没有进行登录校验，服务端在收到消息之后仍然会进行消息的处理，这个逻辑其实是有问题的。本章我们学习一下如何使用Pipeline及Handler强大的热插拔机制来实现客户端身份校验。

15.1 身份检验

首先，我们在客户端登录成功之后，标记当前Channel的状态为已登录。

LoginRequestHandler.java

```
protected void channelRead0(ChannelHandlerContext ctx,  
LoginRequestPacket loginRequestPacket)  {  
  
    if (valid(loginRequestPacket))  {  
  
        // ...  
  
        // 基于前面一节的代码，添加如下一行代码  
  
        LoginUtil.markAsLogin(ctx.channel());  
  
    }  
  
    // ...  
  
}
```

LoginUtil.java

```
public static void markAsLogin(Channel channel)  {  
  
    channel.attr(Attributes.LOGIN).set(true);  
  
}
```

在登录成功之后，我们通过给Channel打属性标记的方式，标记这个Channel已成功登录。接下来，我们是不是需要在处理后续的每一种指令前，都判断一下用户是否登录？

LoginUtil.java

```
public static boolean hasLogin(Channel channel) {  
  
    Attribute<Boolean> loginAttr = channel.attr(Attributes.LOGIN);  
  
    return loginAttr.get() != null;  
  
}
```

判断一个用户是否登录很简单，只需要调用LoginUtil.hasLogin(channel)即可。但是，Netty的Pipeline机制帮我们省去了重复添加同一段逻辑的烦恼，我们只需要在后续所有的指令处理Handler之前插入一个用户认证Handler即可。

NettyServer.java

```
.childHandler(new ChannelInitializer<NioSocketChannel>() {  
  
    protected void initChannel(NioSocketChannel ch) {  
  
        ch.pipeline().addLast(new PacketDecoder());  
  
        ch.pipeline().addLast(new LoginRequestHandler());  
  
        // 新增加用户认证handler ch.pipeline().addLast(new AuthHandler());  
  
        ch.pipeline().addLast(new MessageRequestHandler());  
  
        ch.pipeline().addLast(new PacketEncoder());  
  
    }  
  
});
```

从上面的代码可以看出，我们在MessageRequestHandler之前插入了一个AuthHandler，因此MessageRequestHandler以及后续所有与指令相关的Handler（后面小节会逐个添加）的处理都会经过AuthHandler的一层过滤，只要在AuthHandler里

处理完与身份认证相关的逻辑，后续所有的Handler都不用再操心身份认证这个逻辑，我们来看AuthHandler的具体实现。

AuthHandler.java

```
public class AuthHandler extends ChannelInboundHandlerAdapter {  
  
    @Override  
  
    public void channelRead(ChannelHandlerContext ctx, Object msg)  
throws Exception {  
  
    if (!LoginUtil.hasLogin(ctx.channel())) {  
  
        ctx.channel().close();  
  
    } else {  
  
        super.channelRead(ctx, msg);  
  
    }  
}  
}
```

1. AuthHandler继承自ChannelInboundHandlerAdapter，覆盖了channelRead()方法，表明它可以处理所有类型的数据。

2. 在channelRead()方法里，在决定是否把读到的数据传递到后续指令处理器之前，首先会判断是否登录成功。如果未登录，则直接强制关闭连接，否则，就把读到的数据向下传递，传递给后续指令处理器。

AuthHandler的处理逻辑其实就这么简单。但是，有的读者可能要问，如果客户端已经登录成功，那么在每次处理客户端数据之前，都要经历这么一段逻辑。比如，平均每次用户登录之后发送100次消息，其实剩余的99次身份校验逻辑都是没有必要的，因为只要连接未断开，只要客户端成功登录过，后续就不需要再进行客户端的身份校验。

这里我们为了演示，身份认证逻辑比较简单，在实际生产环境中，身份认证逻辑可能会更复杂。我们需要寻找一种途径来避免资源与性能的浪费，使用ChannelHandler的

热插拔机制完全可以做到这一点。

15.2 移除校验逻辑

对于Netty的设计来说，Handler其实可以看作一段功能相对聚合的逻辑，然后通过Pipeline把一个个小的逻辑聚合起来，串成一个功能完整的逻辑链。既然可以把逻辑串起来，就可以做到动态删除一个或多个逻辑。

在客户端校验通过之后，我们不再需要AuthHandler这段逻辑，而删除这段逻辑只需要一行代码即可实现。

AuthHandler.java

```
public class AuthHandler extends ChannelInboundHandlerAdapter {

    @Override

    public void channelRead(ChannelHandlerContext ctx, Object msg)
        throws Exception {
        if (!LoginUtil.hasLogin(ctx.channel())) {
            ctx.channel().close();
        } else {
            // 一行代码实现逻辑的删除
            ctx.pipeline().remove(this);
            super.channelRead(ctx, msg);
        }
    }

    @Override

    public void handlerRemoved(ChannelHandlerContext ctx) {
        if (LoginUtil.hasLogin(ctx.channel())) {
```

```

        System.out.println("当前连接登录验证完毕，无须再次验证，AuthHandler 被移
除");
    } else {
        System.out.println("无登录验证，强制关闭连接！");
    }
}
}
}

```

在上面的代码中，判断如果已经经过权限认证，那么就直接调用Pipeline的remove()方法删除自身，这里的this指的其实就是AuthHandler这个对象。删除之后，这条客户端连接的逻辑链中就不再有这段逻辑了。

另外，我们覆盖了handlerRemoved()方法，主要用于后续演示部分的内容。接下来，我们进行实际演示。

15.3 身份校验演示

在演示之前，对于客户端侧的代码，在客户端向服务端发送消息的逻辑中，我们先把每次都判断是否登录的逻辑去掉，这样就可以在客户端未登录的情况下向服务端发送消息。

NettyClient.java

```

private static void startConsoleThread(Channel channel) {
    new Thread(() -> {
        while (!Thread.interrupted()) {
            // 这里注释掉
            // if (LoginUtil.hasLogin(channel)) {
            System.out.println("输入消息发送至服务端：");
            Scanner sc = new Scanner(System.in);

```

```

        String line = sc.nextLine();

        channel.writeAndFlush(new MessageRequestPacket(line));

    // }

}

} ).start();

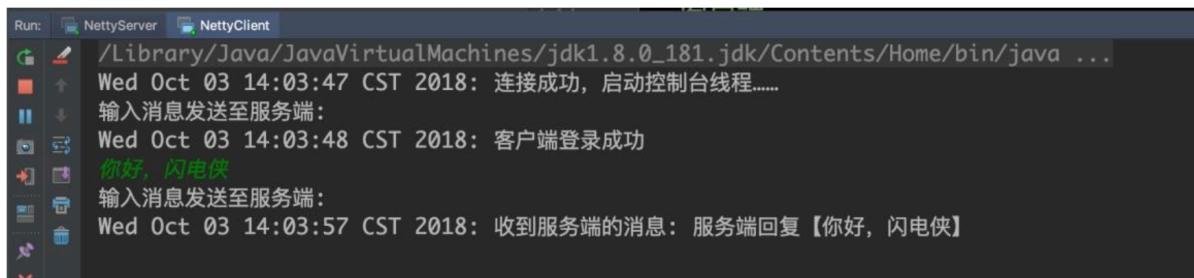
}

```

15.3.1 有身份认证的演示

我们先启动服务端，再启动客户端。在客户端的控制台，我们输入消息发送至服务端，此时客户端与服务端控制台的输出分别如下面两图所示。

客户端



The screenshot shows the Java application's terminal window with the following log output:

```

Run: NettyServer NettyClient
/Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/bin/java ...
Wed Oct 03 14:03:47 CST 2018: 连接成功, 启动控制台线程.....
输入消息发送至服务端:
Wed Oct 03 14:03:48 CST 2018: 客户端登录成功
你好, 闪电侠
输入消息发送至服务端:
Wed Oct 03 14:03:57 CST 2018: 收到服务端的消息: 服务端回复【你好, 闪电侠】

```

服务端



The screenshot shows the Java application's terminal window with the following log output:

```

Run: NettyServer NettyClient
/Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/bin/java ...
Wed Oct 03 14:03:44 CST 2018: 端口[8000]绑定成功!
Wed Oct 03 14:03:48 CST 2018: 收到客户端登录请求....
Wed Oct 03 14:03:48 CST 2018: 登录成功!
当前连接登录验证完毕, 无须再次验证, AuthHandler 被移除
Wed Oct 03 14:03:57 CST 2018: 收到客户端消息: 你好, 闪电侠

```

观察服务端侧的控制台，我们可以看到，在客户端第一次发来消息的时候，AuthHandler判断当前用户已通过身份认证，直接移除自身。移除之后，回调 handlerRemoved()方法，这块内容也是上章ChannelHandler生命周期的一部分。

15.3.2 无身份认证的演示

接下来，我们演示一下客户端在未登录的情况下如何发送消息到服务端。我们在 LoginResponse-Handler 中删除发送登录指令的逻辑。

LoginResponseHandler.java

```
public class LoginResponseHandler extends SimpleChannelInboundHandler<LoginResponsePacket> {

    @Override

    public void channelActive(ChannelHandlerContext ctx) {

        // 创建登录对象

        LoginRequestPacket loginRequestPacket = new LoginRequestPacket();

        loginRequestPacket.setUserId(UUID.randomUUID().toString());

        loginRequestPacket.setUsername("flash");

        loginRequestPacket.setPassword("pwd");

        // 删除登录的逻辑

        // ctx.channel().writeAndFlush(loginRequestPacket);

    }

    @Override

    public void channelInactive(ChannelHandlerContext ctx) {

        System.out.println("客户端连接被关闭! ");

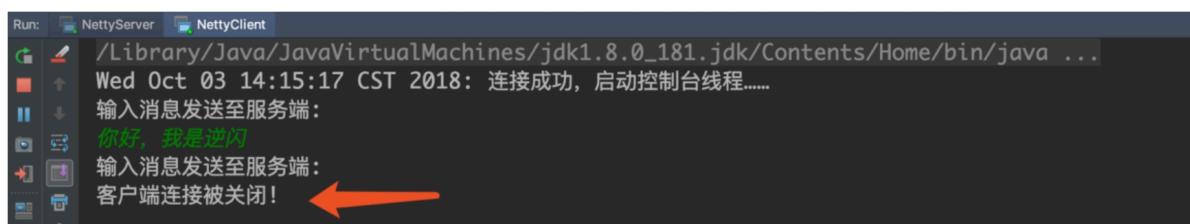
    }

}
```

我们把客户端向服务端写登录指令的逻辑删除，然后覆盖 channelInactive() 方法，用于验证客户端连接是否会被关闭。

接下来，我们先运行服务端，再运行客户端，并且在客户端的控制台输入文本之后发送给服务端。此时客户端与服务端控制台的输出分别如下面两图所示。

客户端



Run: NettyServer NettyClient
/Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/bin/java ...
Wed Oct 03 14:15:17 CST 2018: 连接成功, 启动控制台线程.....
输入消息发送至服务端:
你好, 我是逆闪
输入消息发送至服务端:
客户端连接被关闭!

An orange arrow points to the last line of text: "客户端连接被关闭!".

服务端



Run: NettyServer NettyClient
/Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/bin/java ...
Wed Oct 03 14:15:15 CST 2018: 端口[8000]绑定成功!
无登录验证, 强制关闭连接!

An orange arrow points to the last line of text: "无登录验证, 强制关闭连接!".

由此看到，如果客户端第一个指令为非登录指令，则AuthHandler直接将客户端连接关闭，并且从有关ChannelHandler生命周期的内容中也可以看到，服务端侧的handlerRemoved()方法和客户端侧代码的channelInActive()会被回调到。

15.4 总结

1. 如果有很多业务逻辑的Handler都要进行某些相同的操作，则我们完全可以抽取出一个Handler来单独处理。
2. 如果某一个独立的逻辑在执行几次之后（这里是一次）不需要再执行，则可以通过ChannelHandler的热插拔机制来实现动态删除逻辑，使应用程序的性能处理更为高效。

15.5 思考

在最后一部分的演示中，对于客户端在登录情况下发送消息以及在未登录情况下发送消息，AuthHandler的其他回调方法分别是如何执行的，为什么？

第16章

客户端互聊的原理与实现

本章我们来实现客户端互聊的逻辑，我们先来看一下本章学完之后，单聊的效果是什么样的。

16.1 最终效果

下面我们来看看单聊的最终效果。

服务端



```
Run: NettyServer NettyClient NettyClient
/Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/bin/java ...
Thu Oct 04 10:26:06 CST 2018: 端口[8000]绑定成功!
[闪电侠]登录成功
[逆闪]登录成功
```

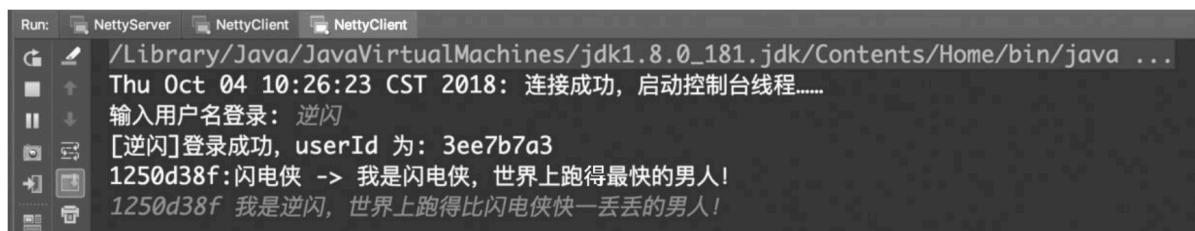
服务端启动之后，两个客户端陆续登录。

客户端1



```
Run: NettyServer NettyClient NettyClient
/Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/bin/java ...
Thu Oct 04 10:26:09 CST 2018: 连接成功, 启动控制台线程.....
输入用户名登录: 闪电侠
[闪电侠]登录成功, userId 为: 1250d38f
3ee7b7a3 我是闪电侠, 世界上跑得最快的男人!
3ee7b7a3:逆闪 -> 我是逆闪, 世界上跑得比闪电侠快一丢丢的男人!
```

客户端2



```
Run: NettyServer NettyClient NettyClient
/Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/bin/java ...
Thu Oct 04 10:26:23 CST 2018: 连接成功, 启动控制台线程.....
输入用户名登录: 逆闪
[逆闪]登录成功, userId 为: 3ee7b7a3
1250d38f:闪电侠 -> 我是闪电侠, 世界上跑得最快的男人!
1250d38f 我是逆闪, 世界上跑得比闪电侠快一丢丢的男人!
```

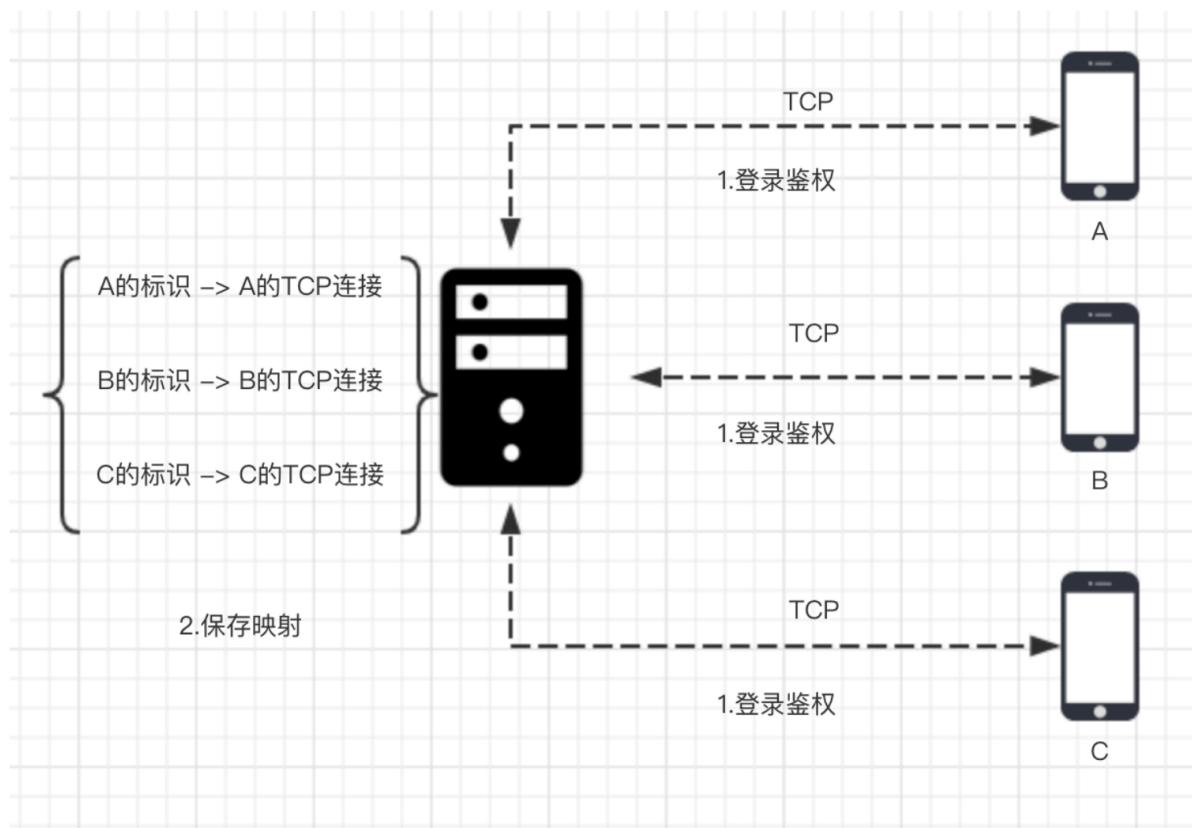
1. 客户端启动之后，我们在控制台输入用户名，服务端随机分配一个userId给客户端，这里我们省去了通过账号、密码注册的过程，userId就在服务端随机生成了，生产环境中可能会持久化在数据库，然后每次通过账号、密码去“捞”。

2. 当有两个客户端登录成功之后，在控制台输入userId+空格+消息，这里的userId是消息接收方的标识，消息接收方的控制台接着就会显示另外一个客户端发来的消息。

一对一单聊的核心逻辑其实就这么简单，稍加改动就可以用在生产环境中。下面我们就来一起学习如何实现一对一单聊。

16.2 一对一单聊的原理

一对一单聊的原理在前面的章节已经介绍过，我们再来重温一下，如下图所示。



1. A要和B聊天，首先A和B需要与服务器建立连接，然后进行一次登录流程，服务端保存用户标识和TCP连接的映射关系。

2. A发消息给B，首先需要将带有B标识的消息数据包发送到服务器，然后服务器从消息数据包中获得B的标识，找到对应B的连接，将消息发送给B。

掌握原理之后，我们就来逐个实现其中的逻辑。

16.3 一对单职业的实现

16.3.1 用户登录状态与Channel的绑定

我们来看服务端在单聊中是如何处理登录消息的。

LoginRequestHandler.java

```
// 这里略去了非关键部分的代码

protected void channelRead0(ChannelHandlerContext ctx,
LoginRequestPacket
loginRequestPacket) {

    LoginResponsePacket loginResponsePacket = xxx;

    String userId = randomUserId();

    loginResponsePacket.setUserId(userId);

    SessionUtil.bindSession(new Session(userId,
loginRequestPacket.getUserName(),

ctx.channel()));

    // 登录响应

    ctx.channel().writeAndFlush(loginResponsePacket);

}

// 用户断线之后取消绑定

public void channelInactive(ChannelHandlerContext ctx) {

    SessionUtil.unbindSession(ctx.channel());

}
```

登录成功之后，服务端首先创建一个Session对象，表示用户当前的会话信息。在这个应用程序里面，Session只有下面两个字段。

Session.java

```
public class Session {  
  
    // 用户唯一性标识  
  
    private String userId;  
  
    private String userName;  
  
}
```

在实际生产环境中，Session中的字段可能较多，比如头像URL、年龄、性别等。

然后我们调用SessionUtil.bindSession()保存用户的会话信息，具体实现如下。

SessionUtil.java

```
public class SessionUtil {  
  
    // userId -> channel 的映射  
  
    private static final Map<String, Channel> userIdChannelMap = new  
    ConcurrentHashMap<>();  
  
    public static void bindSession(Session session, Channel channel) {  
  
        userIdChannelMap.put(session.getUserId(), channel);  
  
        channel.attr(Attributes.SESSION).set(session);  
  
    }  
  
    public static void unBindSession(Channel channel) {  
  
        if (hasLogin(channel)) {  
  
            userIdChannelMap.remove(getSession(channel).getUserId());  
        }  
    }  
}
```

```
channel.attr(Attributes.SESSION).set(null);

    }

}

public static boolean hasLogin(Channel channel) {

    return channel.hasAttr(Attributes.SESSION);

}

public static Session getSession(Channel channel) {

    return channel.attr(Attributes.SESSION).get();

}

public static Channel getChannel(String userId) {

    return userIdChannelMap.get(userId);

}

}
```

1.SessionUtil里维持了一个userId->Channel的映射Map，调用bindSession()方法的时候，在Map里保存这个映射关系。SessionUtil还提供了getChannel()方法，这样就可以通过userId获得对应的Channel。

2.除了在Map里维持映射关系，在bindSession()方法中，我们还给Channel附上了一个属性，这个属性就是当前用户的Session。我们也提供了getSession()方法，非常方便地获得对应Channel的会话信息。

3.这里的SessionUtil其实就是第15.1节的LoginUtil，这里进行了重构，其中hasLogin()方法，只需要判断当前是否有用户的会话信息即可。

4.在LoginRequestHandler中，我们还重写了channelInactive()方法。用户下线之后，我们需要在内存里自动删除userId到Channel的映射关系，这是通过调用SessionUtil.unBindSession()来实现的。

关于保存用户会话信息的逻辑其实就这么，总结一下就是：登录的时候保存会话信息，登出的时候删除会话信息。下面我们就来实现服务端接收消息并转发的逻辑。

16.3.2 服务端接收消息并转发的实现

我们重新定义一下客户端发送给服务端的消息的数据包格式。

MessageRequestPacket.java

```
public class MessageRequestPacket extends Packet {  
  
    private String toUserId;  
  
    private String message;  
  
}
```

数据包格式很简单，toUserId表示要发送给哪个用户，message表示具体内容。我们来看一下服务端的消息处理Handler是如何处理消息的。

MessageRequestHandler.java

```
public class MessageRequestHandler extends  
SimpleChannelInboundHandler<MessageRequestPacket> {  
  
    @Override  
  
    protected void channelRead0(ChannelHandlerContext ctx,  
        MessageRequestPacket  
  
        messageRequestPacket) {  
  
        // 1.获得消息发送方的会话信息  
  
        Session session = SessionUtil.getSession(ctx.channel());  
  
        // 2.通过消息发送方的会话信息构造要发送的消息  
  
        MessageResponsePacket messageResponsePacket = new  
        MessageResponsePacket();  
  
        messageResponsePacket.setFromUserId(session.getUserId());
```

```
messageResponsePacket.setFromUserName(session.getUserName());  
  
messageResponsePacket.setMessage(messageRequestPacket.getMessage());  
  
// 3.获得消息接收方的Channel  
  
Channel toUserChannel =  
SessionUtil.getChannel(messageRequestPacket.getToUserId());  
  
// 4.将消息发送给消息接收方  
  
if (toUserChannel != null && SessionUtil.hasLogin(toUserChannel)) {  
  
    toUserChannel.writeAndFlush(messageResponsePacket);  
  
} else {  
  
    System.err.println(" [" + messageRequestPacket.getToUserId() + "] 不  
在线，发  
送失败! ");  
  
}  
  
}  
  
}  
  
}
```

1.服务端在收到客户端发来的消息之后，首先获得当前用户也就是消息发送方的会话信息。

2.获得消息发送方的会话信息之后，构造一个发送给客户端的消息对象 MessageResponsePacket，填上消息发送方的用户标识、昵称、消息内容。

3.通过消息接收方的标识获得对应的Channel。

4.如果消息接收方当前是登录状态，则直接发送；如果不在线，则控制台打印一条警告消息。

这里，服务端的功能相当于消息转发：收到一个客户端的消息之后，构建一条发送给另一个客户端的消息，接着获得另一个客户端的Channel，然后通过writeAndFlush()

写出来。我们再来看一下客户端收到消息之后的处理逻辑。

16.3.3 客户端接收消息的逻辑处理

MessageResponseHandler.java

```
public class MessageResponseHandler extends  
SimpleChannelInboundHandler<MessageResponsePacket> {  
  
    @Override  
  
    protected void channelRead0(ChannelHandlerContext ctx,  
MessageResponsePacket  
  
messageResponsePacket) {  
  
    String fromUserId = messageResponsePacket.getFromUserId();  
  
    String fromUserName = messageResponsePacket.getFromUserName();  
  
    System.out.println(fromUserId + ":" + fromUserName + " -> " +  
  
messageResponsePacket.getMessage());  
  
}  
  
}
```

客户端收到消息之后，只是把当前消息打印出来。这里把发送方的用户标识打印出来是为了方便我们在控制台回消息的时候，可以直接复制。到这里，所有的核心逻辑其实已经完成了，我们还差最后一环：在客户端控制台进行登录和发送消息的逻辑。

16.3.4 客户端控制台登录和发送消息

我们回到客户端的启动类，改造一下控制台的逻辑。

NettyClient.java

```
private static void startConsoleThread(Channel channel) {  
  
Scanner sc = new Scanner(System.in);
```

```
 LoginRequestPacket loginRequestPacket = new LoginRequestPacket();

new Thread(() -> {

    while (! Thread.interrupted()) {

        if (! SessionUtil.hasLogin(channel)) {

            System.out.print("输入用户名登录:  ");

            String username = sc.nextLine();

            loginRequestPacket.setUserName(username);

            // 使用默认的密码

            loginRequestPacket.setPassword("pwd");

            // 发送登录数据包

            channel.writeAndFlush(loginRequestPacket);

            waitForLoginResponse();

        } else {

            String toUserId = sc.next();

            String message = sc.next();

            channel.writeAndFlush(new MessageRequestPacket(toUserId, message));

        }

    }

} ).start();

}

private static void waitForLoginResponse() {
```

```
try  {

    Thread.sleep(1000);

} catch (InterruptedException ignored)  {

}

}
```

在客户端启动的时候，起一个线程：

- 1.如果当前用户还未登录，我们在控制台输入一个用户名，然后构造一个登录数据包发送给服务器。发完之后，等待一个超时时间，可以当作登录逻辑的最大处理时间。
- 2.如果当前用户已经是登录状态，我们可以在控制台输入消息接收方的userId，然后输入一个空格，再输入消息的具体内容，这样我们就可以构建一个消息数据包，发送到服务端。

关于单聊的原理和实现到这里就讲解完成了，最后我们对本章内容做一下总结。

16.4 总结

- 1.我们定义一个会话类Session来维持用户的登录信息，用户登录的时候绑定Session与Channel，用户登出或者断线的时候解绑Session与Channel。
- 2.服务端处理消息的时候，通过消息接收方的标识，获得消息接收方的Channel，调用writeAndFlush()方法将消息发送给消息接收方。

16.5 思考

本章其实还少了用户登出请求和响应的指令处理，你能否说出，对于登出指令来说，服务端和客户端分别要做哪些事情？能否自行实现？

第17章

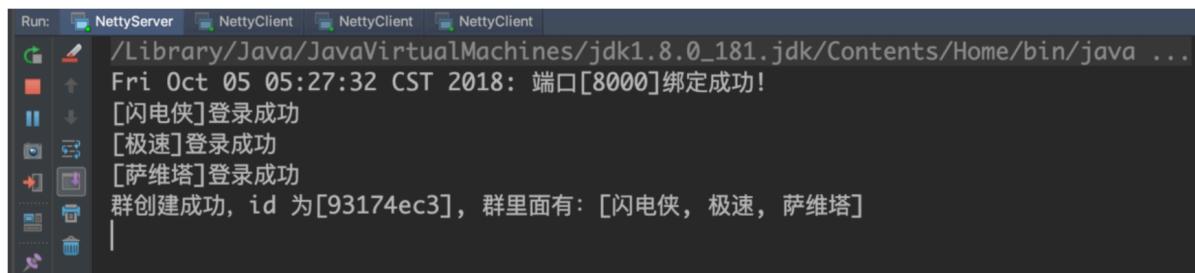
群聊的发起与通知

本章我们学习如何创建一个群聊，并通知群聊中的各位成员。我们依然是先来看下最终效果是什么样的。

17.1 最终效果

群聊的最终效果如下。

服务端



```
Run: NettyServer NettyClient NettyClient NettyClient
/ Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/bin/java ...
Fri Oct 05 05:27:32 CST 2018: 端口[8000]绑定成功!
[闪电侠]登录成功
[极速]登录成功
[萨维塔]登录成功
群创建成功, id 为[93174ec3], 群里面有: [闪电侠, 极速, 萨维塔]
```

创建群聊的客户端



```
Run: NettyServer NettyClient NettyClient NettyClient
/ Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/bin/java ...
Fri Oct 05 05:27:40 CST 2018: 连接成功, 启动控制台线程.....
输入用户名登录: 闪电侠
[闪电侠]登录成功, userId 为: 1767a6bb
createGroup
【拉人群聊】输入 userId 列表, userId 之间英文逗号隔开: 1767a6bb,0f197a37,b0045b0e
群创建成功, id 为[93174ec3], 群里面有: [闪电侠, 极速, 萨维塔]
```

其他客户端

```
Run: NettyServer NettyClient NettyClient NettyClient
/Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/bin/java ...
Fri Oct 05 05:27:50 CST 2018: 连接成功, 启动控制台线程.....
输入用户名登录: 极速
[极速] 登录成功, userId 为: 0f197a37
群创建成功, id 为[93174ec3], 群里面有: [闪电侠, 极速, 萨维塔]
```

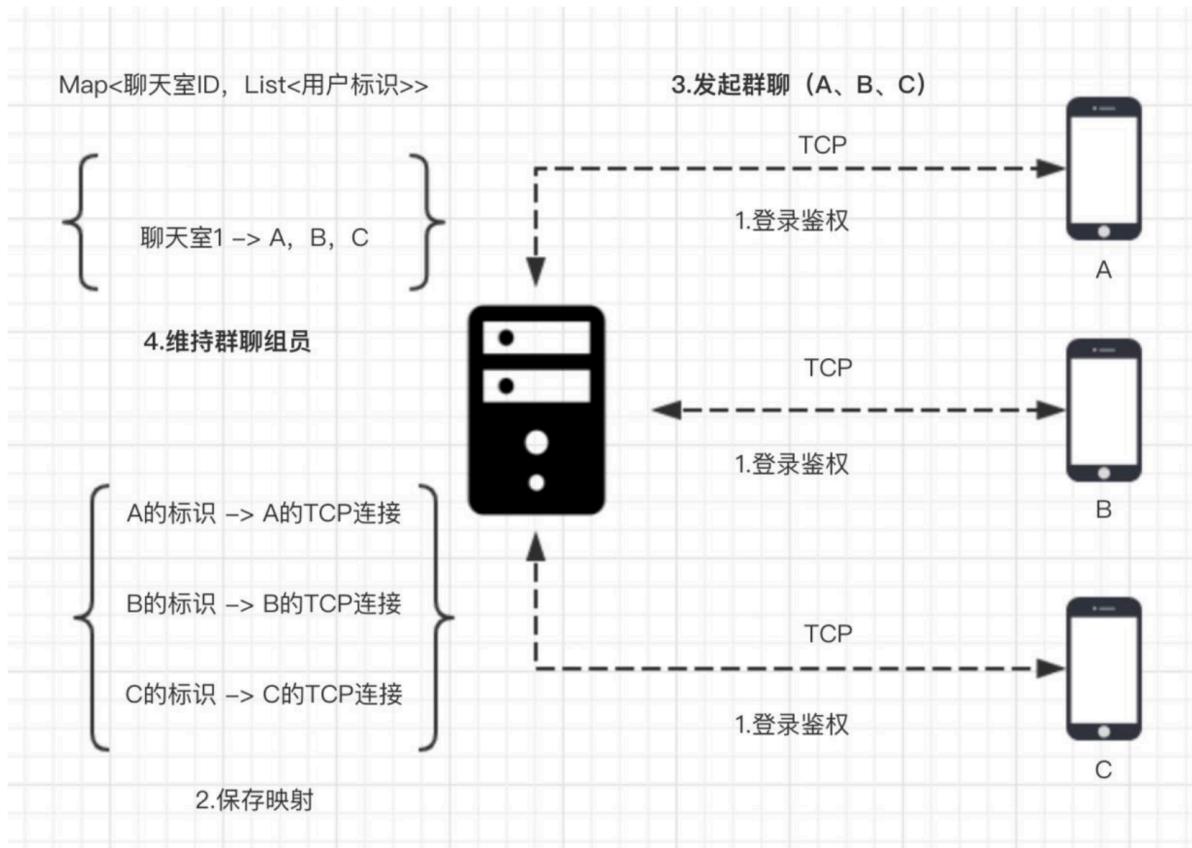
```
Run: NettyServer NettyClient NettyClient NettyClient
/Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/bin/java ...
Fri Oct 05 05:28:07 CST 2018: 连接成功, 启动控制台线程.....
输入用户名登录: 萨维塔
[萨维塔] 登录成功, userId 为: b0045b0e
群创建成功, id 为[93174ec3], 群里面有: [闪电侠, 极速, 萨维塔]
```

- 1.依然是三位用户依次登录服务器，分别是闪电侠、极速、萨维塔。
- 2.我们在闪电侠的控制台输入createGroup指令，提示创建群聊需要输入userId列表，然后我们输入以英文逗号分隔的userId。
- 3.群聊创建成功之后，分别在服务端和三个客户端弹出提示消息，包括群ID及群里各位用户的昵称。

17.2 群聊的原理

关于群聊的原理，我们在即时聊天系统简介中已经学习过，现在再来重温一下。

群聊指的是一个组内多位用户之间的聊天，一位用户发到群组的消息会被组内任何一个成员接收。下面来看群聊的基本流程，如下图所示。



群聊的基本流程其实和单聊类似。

1.A、B、C依然会经历登录流程，服务端保存用户标识对应的TCP连接。

2.A发起群聊的时候，将A、B、C的标识发送至服务端，服务端拿到之后建立一个群ID，然后把这个ID与A、B、C的标识绑定。

3.群聊里的任意一方在群里聊天的时候，将群ID发送至服务端，服务端获得群ID之后，取出对应的用户标识，遍历用户标识对应的TCP连接，就可以将消息发送至每一个群聊成员。

这一章，我们把重点放在创建一个群聊上，由于控制台输入的指令越来越多，因此在正式开始之前，我们先对控制台程序稍作重构。

17.3 控制台程序重构

17.3.1 创建控制台命令执行器

首先，把在控制台要执行的操作抽象出来，抽象出一个接口。

ConsoleCommand.java

```
public interface ConsoleCommand {  
  
    void exec(Scanner scanner, Channel channel);  
  
}
```

17.3.2 管理控制台命令执行器

接着，创建一个管理类来对这些操作进行管理。

ConsoleCommandManager.java

```
public class ConsoleCommandManager implements ConsoleCommand {  
  
    private Map<String, ConsoleCommand> consoleCommandMap;  
  
    public ConsoleCommandManager() {  
  
        consoleCommandMap = new HashMap<>();  
  
        consoleCommandMap.put("sendToUser", new SendToUserConsoleCommand());  
  
        consoleCommandMap.put("logout", new LogoutConsoleCommand());  
  
        consoleCommandMap.put("createGroup", new  
CreateGroupConsoleCommand());  
  
    }  
  
    @Override  
  
    public void exec(Scanner scanner, Channel channel) {  
  
        // 获取第一个指令  
  
        String command = scanner.next();  
  
        ConsoleCommand consoleCommand = consoleCommandMap.get(command);  
  
        if (consoleCommand != null) {  
    }
```

```
        consoleCommand.exec(scanner, channel);

    } else {

        System.err.println("无法识别 [" + command + "] 指令, 请重新输入! ");

    }

}

}

}
```

1.在这个管理类中，把所有要管理的控制台指令都放到一个Map中。

2.执行具体操作的时候，先获取控制台第一个输入的指令，这里以字符串代替比较清晰（这里我们已经实现了第16章思考题中的登出操作），然后通过这个指令拿到对应的控制台命令执行器执行。

这里我们就以创建群聊为例：首先在控制台输入createGroup，然后按下回车键，就会进入CreateGroupConsoleCommand这个类进行处理。

CreateGroupConsoleCommand.java

```
public class CreateGroupConsoleCommand implements ConsoleCommand {

    private static final String USER_ID_SPLITTER = ", ";

    @Override

    public void exec(Scanner scanner, Channel channel) {

        CreateGroupRequestPacket createGroupRequestPacket = new
CreateGroupRequestPacket();

        System.out.print("【拉人群聊】输入userId列表, userId之间英文逗号隔开: ");

        String userIds = scanner.next();

        createGroupRequestPacket.setUserIdList(Arrays.asList(userIds.split
(USER_ID_SPLITTER)));
    }
}
```

```
channel.writeAndFlush(createGroupRequestPacket);

    }

}
```

进入CreateGroupConsoleCommand的逻辑之后，我们创建了一个群聊创建请求的数据包，然后提示输入以英文逗号分隔的userId的列表。填充完这个数据包之后，调用writeAndFlush()方法就可以发送创建群聊的指令到服务端。

最后来看经过改造的与客户端控制台线程相关的代码。

NettyClient.java

```
private static void startConsoleThread(Channel channel) {

    ConsoleCommandManager consoleCommandManager = new
ConsoleCommandManager();

    LoginConsoleCommand loginConsoleCommand = new LoginConsoleCommand();

    Scanner scanner = new Scanner(System.in);

    new Thread(() -> {

        while (! Thread.interrupted()) {

            if (! SessionUtil.hasLogin(channel)) {

                loginConsoleCommand.exec(scanner, channel);

            } else {

                consoleCommandManager.exec(scanner, channel);

            }
        }
    })
}
```

```
    } ).start();  
  
}
```

抽取出控制台指令执行器之后，客户端控制台的逻辑已经相比之前清晰很多了，可以非常方便地在控制台模拟各种在IM聊天窗口的操作。接下来，我们看一下如何创建群聊。

17.4 创建群聊的实现

17.4.1 客户端发送创建群聊请求

通过前面讲述控制台逻辑的重构，我们已经了解到，我们发送了一个 CreateGroupRequestPacket 数据包到服务端，这个数据包的格式如下。

CreateGroupRequestPacket.java

```
public class CreateGroupRequestPacket extends Packet {  
  
    private List<String> userIdList;  
  
}
```

它只包含了一个列表，这个列表就是需要拉取群聊的用户列表。我们来看下服务端是如何处理的。

17.4.2 服务端处理创建群聊请求

我们依然创建一个Handler来处理新的指令。

NettyServer.java

```
.childHandler(new ChannelInitializer<NioSocketChannel>() {  
  
    protected void initChannel(NioSocketChannel ch) {  
  
        // ...  
  
        // 添加一个 handler  
  
        ch.pipeline().addLast(new CreateGroupRequestHandler());
```

```
// ...
}

} );
```

我们来看一下这个Handler具体做哪些事情。

CreateGroupRequestHandler.java

```
public class CreateGroupRequestHandler extends
SimpleChannelInboundHandler<CreateGroupRequestPacket> {

@Override

protected void channelRead0(ChannelHandlerContext ctx,
CreateGroupRequestPacket

createGroupRequestPacket) {

List<String> userIdList = createGroupRequestPacket.getUserIdList();

List<String> userNameList = new ArrayList<>();

// 1. 创建一个channel分组

ChannelGroup channelGroup = new DefaultChannelGroup(ctx.executor());

// 2. 筛选出待加入群聊的用户的channel和userName

for (String userId : userIdList) {

Channel channel = SessionUtil.getChannel(userId);

if (channel != null) {

channelGroup.add(channel);
}
}
}
}
```

```
userNameList.add(SessionUtil.getSession(channel).getUserName());  
}  
}  
  
// 3. 创建群聊创建结果的响应  
  
CreateGroupResponsePacket createGroupResponsePacket = new  
CreateGroupResponsePacket();  
  
createGroupResponsePacket.setSuccess(true);  
  
createGroupResponsePacket.setGroupId(IDUtil.randomUUID());  
  
createGroupResponsePacket.setUserNameList(userNameList);  
  
// 4. 给每个客户端都发送拉群通知  
  
channelGroup.writeAndFlush(createGroupResponsePacket);  
  
System.out.print("群创建成功，id 为 [" +  
createGroupResponsePacket.getGroupId() +  
"] , ");  
  
System.out.println("群里面有: " +  
createGroupResponsePacket.getUserNameList());  
}  
}
```

整个过程可以分为以下4个步骤。

1. 创建一个ChannelGroup。这里简单介绍一下ChannelGroup：它可以把多个Channel的操作聚合在一起，可以往它里面添加、删除Channel，也可以进行Channel的批量读写、关闭等操作，详细的功能读者可以自行查阅这个接口的方法。这里的一个群组其实就是一个Channel的分组集合，使用ChannelGroup非常方便。

2.遍历待加入群聊的userId，如果存在该用户，就把对应的Channel添加到ChannelGroup中，用户昵称也被添加到昵称列表中。

3.创建一个创建群聊响应的对象，其中groupId是随机生成的，群聊创建结果共有三个字段，这里就不展开对这个类进行说明了。

4.调用ChannelGroup的聚合发送功能，将拉群的通知批量地发送到客户端，接着在服务端控制台打印创建群聊成功的信息。至此，服务端处理创建群聊请求的逻辑结束。

我们再来看客户端处理创建群聊响应。

17.4.3 客户端处理创建群聊响应

首先，客户端依然创建一个Handler来处理新的指令。

NettyClient.java

```
.handler(new ChannelInitializer<SocketChannel>() {
    @Override
    public void initChannel(SocketChannel ch) {
        // ...
        // 添加一个新的Handler来处理创建群聊成功响应的指令
        ch.pipeline().addLast(new CreateGroupResponseHandler());
        // ...
    }
});
```

然后，在应用程序里，我们仅仅把创建群聊成功之后的具体信息打印出来。

CreateGroupResponseHandler.java

```
public class CreateGroupResponseHandler extends
SimpleChannelInboundHandler<CreateGroupResponsePacket> {
}
```

```
    @Override

    protected void channelRead0(ChannelHandlerContext ctx,
CreateGroupResponsePacket

createGroupResponsePacket)  {

    System.out.print("群创建成功， id 为 [ " +
createGroupResponsePacket.getGroupId() +

"] , ");

    System.out.println("群里面有: " +
createGroupResponsePacket.getUserNameList());

}

}
```

在实际生产环境中，CreateGroupResponsePacket对象里可能有更多信息，以上逻辑的处理也会更加复杂，不过这里已经能说明问题了。

17.5 总结

- 1.群聊的原理和单聊类似，都是通过标识拿到Channel。
- 2.重构了控制台的程序结构，在实际带有UI的IM应用中，我们输入的第一个指令其实就是对应我们点击UI的某些按钮或菜单的操作。
- 3.通过ChannelGroup，可以很方便地对一组Channel进行批量操作。

17.6 思考

如何实现在某个客户端拉取群聊成员的时候，不需要输入自己的用户ID，并且展示创建群聊消息的时候，不显示自己的昵称？

第18章

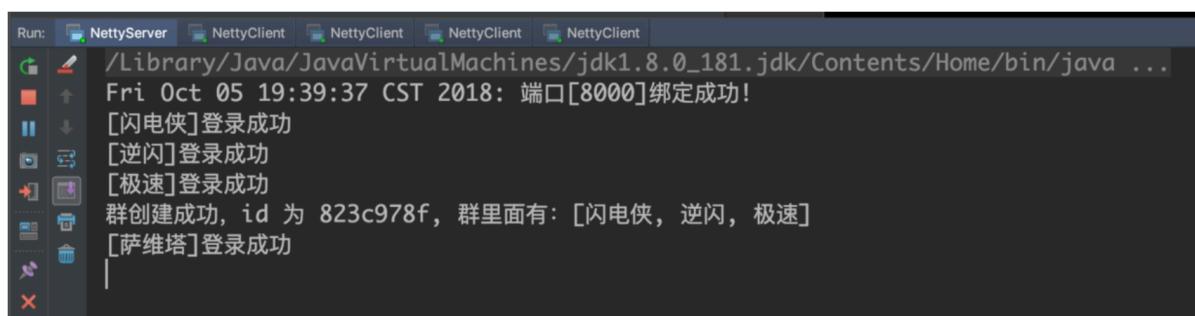
群聊的成员管理

在上一章中，我们已经学习了如何创建群聊并通知群聊的各位成员。本章中我们来实现群成员管理，包括群的加入、退出和获取群成员列表等功能。有了前面两章的基础，相信本章的内容对读者来说会比较简单。在开始之前，我们依然先来看一下最终效果。

18.1 最终效果

群成员管理的最终效果如下图所示。

服务端



The screenshot shows the NettyServer application's log window. The log output is as follows:

```
Run: NettyServer NettyClient NettyClient NettyClient NettyClient
/Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/bin/java ...
Fri Oct 05 19:39:37 CST 2018: 端口[8000]绑定成功!
[闪电侠]登录成功
[逆闪]登录成功
[极速]登录成功
群创建成功, id 为 823c978f, 群里面有: [闪电侠, 逆闪, 极速]
[萨维塔]登录成功
```

从服务端可以看到，闪电侠、逆闪、极速先后登录服务器。随后，闪电侠创建一个群聊。接下来，萨维塔也登录了。这里，我们只展示闪电侠和萨维塔的客户端控制台界面。

客户端（闪电侠）

```
Run: NettyServer NettyClient NettyClient NettyClient NettyClient  
/Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/bin/java ...  
Fri Oct 05 19:39:43 CST 2018: 连接成功, 启动控制台线程.....  
输入用户名登录: 闪电侠  
[闪电侠] 登录成功, userId 为: 57dabe0a  
1.闪电侠邀请逆闪和极速加入群聊  
createGroup  
【拉人群聊】输入 userId 列表, userId 之间英文逗号隔开: 57dabe0a,b73cc4e2,4221d994  
群创建成功, id 为[823c978f], 群里面有: [闪电侠, 逆闪, 极速]  
listGroupMembers  
输入 groupId, 获取群成员列表: 823c978f  
群[823c978f]中的人包括: [4221d994:极速, f75e920f:萨维塔, 57dabe0a:闪电侠, b73cc4e2:逆闪]  
listGroupMembers  
输入 groupId, 获取群成员列表: 823c978f  
群[823c978f]中的人包括: [4221d994:极速, 57dabe0a:闪电侠, b73cc4e2:逆闪]
```

客户端 (萨维塔)

```
Run: NettyServer NettyClient NettyClient NettyClient NettyClient  
/Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/bin/java ...  
Fri Oct 05 19:39:56 CST 2018: 连接成功, 启动控制台线程.....  
输入用户名登录: 萨维塔  
[萨维塔] 登录成功, userId 为: f75e920f  
joinGroup  
输入 groupId, 加入群聊: 823c978f  
加入群[823c978f]成功!  
2.萨维塔加入群聊  
quitGroup  
输入 groupId, 退出群聊: 823c978f  
退出群聊[823c978f]成功!  
4.萨维塔退出群聊
```

我们可以看到四位用户登录成功之后的最终效果。

- 1.闪电侠先拉逆闪和极速加入了群聊，控制台输出群创建成功的消息。
- 2.随后在萨维塔的控制台输入joinGroup之后再输入群ID，加入群聊，控制台显示加入群成功。
- 3.在闪电侠的控制台输入listGroupMembers之后再输入群ID，展示了当前群聊成员包括极速、萨维塔、闪电侠、逆闪。
- 4.在萨维塔的控制台输入quitGroup之后再输入群ID，退出群聊，控制台显示退群成功。
- 5.最后在闪电侠的控制台输入listGroupMembers之后再输入群ID，展示了当前群聊成员中已无萨维塔。

接下来，我们就来实现加入群聊、退出群聊、获取群成员列表三大功能。

18.2 群的加入

18.2.1 在控制台添加群加入命令处理器

JoinGroupConsoleCommand.java

```
public class JoinGroupConsoleCommand implements ConsoleCommand {  
  
    @Override  
  
    public void exec(Scanner scanner, Channel channel) {  
  
        JoinGroupRequestPacket joinGroupRequestPacket = new  
        JoinGroupRequestPacket();  
  
        System.out.print("输入 groupId, 加入群聊: ");  
  
        String groupId = scanner.next();  
  
        joinGroupRequestPacket.setGroupId(groupId);  
  
        channel.writeAndFlush(joinGroupRequestPacket);  
  
    }  
  
}
```

按照前面两章的套路，我们在控制台先添加群加入命令处理器
JoinGroupConsoleCommand。在这个处理器中，我们创建一个指令对象
JoinGroupRequestPacket，填上群ID之后，将数据包发送至服务端。之后，我们将该
控制台指令添加到ConsoleCommandManager。

ConsoleCommandManager.java

```
public class ConsoleCommandManager implements ConsoleCommand {  
  
    public ConsoleCommandManager() {  
  
        // ...  
    }
```

```
consoleCommandMap.put("joinGroup", new JoinGroupConsoleCommand());  
  
// ...  
  
}  
  
}
```

接下来，就轮到服务端来处理加群请求了。

18.2.2 服务端处理加群请求

在服务端的Pipeline中添加对应的Handler—JoinGroupRequestHandler。

NettyServer.java

```
.childHandler(new ChannelInitializer<NioSocketChannel>() {  
  
    protected void initChannel(NioSocketChannel ch) {  
  
        // 添加加群请求处理器  
  
        ch.pipeline().addLast(new JoinGroupRequestHandler());  
  
        // ...  
  
    }  
});
```

JoinGroupRequestHandler的具体逻辑如下。

JoinGroupRequestHandler.java

```
public class JoinGroupRequestHandler extends  
SimpleChannelInboundHandler  
  
<JoinGroupRequestPacket> {  
  
    @Override
```

```
protected void channelRead0(ChannelHandlerContext ctx,
JoinGroupRequestPacket requestPacket) {

    // 1. 获取群对应的ChannelGroup，然后将当前用户的Channel添加进去
    String groupId = requestPacket.getGroupId();
    ChannelGroup channelGroup = SessionUtil.getChannelGroup(groupId);
    channelGroup.add(ctx.channel());

    // 2. 构造加群响应发送给客户端
    JoinGroupResponsePacket responsePacket = new
    JoinGroupResponsePacket();
    responsePacket.setSuccess(true);
    responsePacket.setGroupId(groupId);
    ctx.channel().writeAndFlush(responsePacket);

}

}
```

1.在通过groupId拿到对应的ChannelGroup之后，只需要调用ChannelGroup.add()方法，将加入群聊的用户的Channel添加进去，服务端即完成了加入群聊的逻辑。

2.构造一个加群响应，填入groupId之后，调用writeAndFlush()方法把加群响应发送给加入群聊的客户端。

18.2.3 客户端处理加群响应

我们在客户端的Pipeline中添加对应的Handler—JoinGroupResponseHandler来处理加群之后的响应。

NettyClient.java

```
.handler(new ChannelInitializer<SocketChannel>() {
```

```
    @Override

    public void initChannel(SocketChannel ch)  {

        // 添加加群响应处理器

        ch.pipeline().addLast(new JoinGroupResponseHandler());

        // ...

    }

} );
```

JoinGroupResponseHandler对应的逻辑如下。

JoinGroupResponseHandler.java

```
public class JoinGroupResponseHandler extends
SimpleChannelInboundHandler<JoinGroupResponsePacket> {

    protected void channelRead0(ChannelHandlerContext ctx,
JoinGroupResponsePacket responsePacket)  {

        if (responsePacket.isSuccess())  {

            System.out.println("加入群 [" + responsePacket.getGroupId() + "] 成功!");

        } else {

            System.err.println("加入群 [" + responsePacket.getGroupId() + "] 失败,
原因为:");

    }

}
```

```
" + responsePacket.getReason());  
    }  
}  
}  
}
```

该处理器的逻辑很简单，只是简单地将加群的结果输出到控制台，实际生产环境的IM可能比这要复杂，但是修改起来也非常容易。至此，与加群相关的逻辑就讲解完成了。

18.3 群的退出

关于群的退出逻辑与群的加入逻辑非常类似，这里展示一下关键代码。

服务端退群的核心逻辑为QuitGroupRequestHandler。

QuitGroupRequestHandler.java

```
public class QuitGroupRequestHandler extends  
SimpleChannelInboundHandler<QuitGroupRequestPacket> {  
  
    @Override  
  
    protected void channelRead0(ChannelHandlerContext ctx,  
    QuitGroupRequestPacket  
requestPacket) {  
  
        // 1. 获取群对应的 ChannelGroup，然后将当前用户的Channel移除  
  
        String groupId = requestPacket.getGroupId();  
  
        ChannelGroup channelGroup = SessionUtil.getChannelGroup(groupId);  
  
        channelGroup.remove(ctx.channel());  
  
        // 2. 构造退群响应发送给客户端
```

```
    QuitGroupResponsePacket responsePacket = new
    QuitGroupResponsePacket();

    responsePacket.setGroupId(requestPacket.getGroupId());

    responsePacket.setSuccess(true);

    ctx.channel().writeAndFlush(responsePacket);

}

}
```

从上面的代码可以看到，`QuitGroupRequestHandler`和`JoinGroupRequestHandler`其实是一个逆向的过程。

1.通过`groupId`拿到对应的`ChannelGroup`之后，只需要调用`ChannelGroup.remove()`方法，将当前用户的`Channel`删除，服务端即完成了退群的逻辑。

2.构造一个退群响应，填入`groupId`之后，调用`writeAndFlush()`方法把退群响应发送给退群的客户端。

至此，加群和退群的逻辑就讲解完成了。最后，我们来看一下获取群成员列表的逻辑。

18.4 获取群成员列表

18.4.1 在控制台添加获取群成员列表命令处理器

ListGroupMembersConsoleCommand.java

```
public class ListGroupMembersConsoleCommand implements ConsoleCommand
{

    @Override

    public void exec(Scanner scanner, Channel channel) {

        ListGroupMembersRequestPacket listGroupMembersRequestPacket = new
        ListGroupMembersRequestPacket();
```

```
System.out.print("输入 groupId, 获取群成员列表: ");

String groupId = scanner.next();

listGroupMembersRequestPacket.setGroupId(groupId);

channel.writeAndFlush(listGroupMembersRequestPacket);

}

}
```

依旧按照前面的套路，我们在控制台先添加获取群成员列表命令处理器 ListGroupMembersConsoleCommand。在这个处理器中，我们创建一个指令对象 ListGroupMembersRequestPacket，填上群ID之后，将数据包发送至服务端。之后，将该控制台指令添加到 ConsoleCommandManager。

ConsoleCommandManager.java

```
public class ConsoleCommandManager implements ConsoleCommand {

    public ConsoleCommandManager() {
        // ...
        consoleCommandMap.put("listGroupMembers", new
ListGroupMembersConsoleCommand());
        // ...
    }

}
```

接着，轮到服务端来处理获取群成员列表请求。

18.4.2 服务端处理获取群成员列表请求

在服务端的Pipeline中添加对应的Handler—ListGroupMembersRequestHandler。

NettyServer.java

```
.childHandler(new ChannelInitializer<NioSocketChannel>() {
    protected void initChannel(NioSocketChannel ch) {
        // 添加获取群成员列表请求处理器
        ch.pipeline().addLast(new ListGroupMembersRequestHandler());
        // ...
    }
});
```

ListGroupMembersRequestHandler的具体逻辑如下。

ListGroupMembersRequestHandler.java

```
public class ListGroupMembersRequestHandler extends
SimpleChannelInboundHandler<ListGroupMembersRequestPacket> {
    protected void channelRead0(ChannelHandlerContext ctx,
        JoinGroupRequestPacket requestPacket) {
        // 1. 获取群的ChannelGroup
        String groupId = requestPacket.getGroupId();
        ChannelGroup channelGroup = SessionUtil.getChannelGroup(groupId);
        // 2. 遍历群成员的Channel对应的Session，构造群成员的信息
        List<Session> sessionList = new ArrayList<>();
        for (Channel channel : channelGroup) {
            Session session = SessionUtil.getSession(channel);
```

```
        sessionList.add(session);

    }

// 3. 构建获取群成员列表响应，写回客户端

ListGroupMembersResponsePacket responsePacket = new
ListGroupMembersResponsePacket();

responsePacket.setGroupId(groupId);

responsePacket.setSessionList(sessionList);

ctx.channel().writeAndFlush(responsePacket);

}

}
```

1.通过groupId拿到对应的ChannelGroup。

2.创建一个sessionList用来装载群成员信息，遍历Channel的每个Session，把对应的用户信息都装到sessionList中。在实际生产环境中，这里可能会构造另外一个对象来装载用户信息而非Session。

3.构造一个获取群成员列表的响应指令数据包，填入groupId和群成员信息之后，调用writeAndFlush()方法把响应发送给发起获取群成员列表的客户端。

最后，就剩下客户端来处理获取群成员列表的响应了。

18.4.3 客户端处理获取群成员列表响应

和前面一样，我们在客户端的Pipeline中添加一个Handler—
ListGroupMembersResponseHandler。

NettyClient.java

```
.handler(new ChannelInitializer<SocketChannel>() {
    public void initChannel(SocketChannel ch) {
```

```
// ...

// 添加获取群成员响应处理器

ch.pipeline().addLast(new ListGroupMembersResponseHandler());

// ...

}

} );
```

而ListGroupMembersResponseHandler的逻辑也只是在控制台展示一下群成员的信息。

ListGroupMembersResponseHandler.java

```
public class ListGroupMembersResponseHandler extends
SimpleChannelInboundHandler

<ListGroupMembersResponsePacket> {

    protected void channelRead0(ChannelHandlerContext ctx,
ListGroupMembersResponsePacket
responsePacket) {

    System.out.println("群 [" + responsePacket.getGroupId() + "] 中的人包
括: " +
responsePacket.getSessionList();

    }

}
```

至此，群成员加入、退出，以及获取群成员列表对应的逻辑就全部实现了，其实从本章和前面两章大家可以看到，添加一个新功能是有一定套路的，我们在最后的总结中给出这个套路。

18.5 总结

添加一个服务端和客户端交互的新功能只需要遵循以下步骤。

1. 创建控制台指令对应的ConsoleCommand，并将其添加到ConsoleCommandManager。
2. 在控制台输入指令和数据之后，填入协议对应的指令数据包—xxxRequestPacket，将请求写到服务端。
3. 服务端创建对应的xxxRequestPacketHandler，并将其添加到服务端的Pipeline中，在xxxRequestPacketHandler处理完后构造对应的xxxResponsePacket发送给客户端。
4. 客户端创建对应的xxxResponsePacketHandler，并将其添加到客户端的Pipeline中，最后在xxxResponsePacketHandler中完成响应的处理。
5. 最容易忽略的是，新添加xxxPacket时别忘了完善编解码器PacketCodec中的packetTypeMap。

18.6 思考

1. 实现以下功能：客户端加入或者退出群聊，将加入群聊的消息也通知给群聊中的其他客户端，这个消息需要和发起群聊的客户端区分开，类似“xxx加入群聊yyy”的格式。
2. 实现：当一个群的人数为0的时候，清理内存中与该群相关的信息。

第19章

群聊消息的收发及Netty性能优化

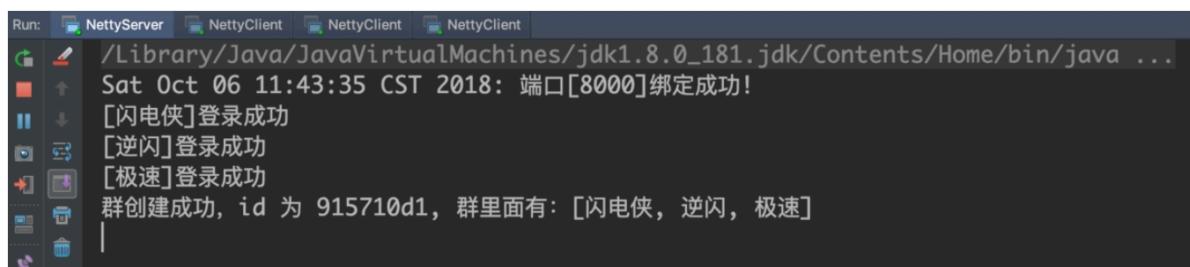
通过第16~18章的学习，相信读者看到本章的标题就已经知道该如何实现本章的功能了。本章在实现了群聊消息收发之后，还会介绍一些与性能优化相关的内容。

开始之前，我们先来看一下群聊消息的最终效果。

19.1 群聊消息的最终效果

群聊消息的最终效果如下图所示。

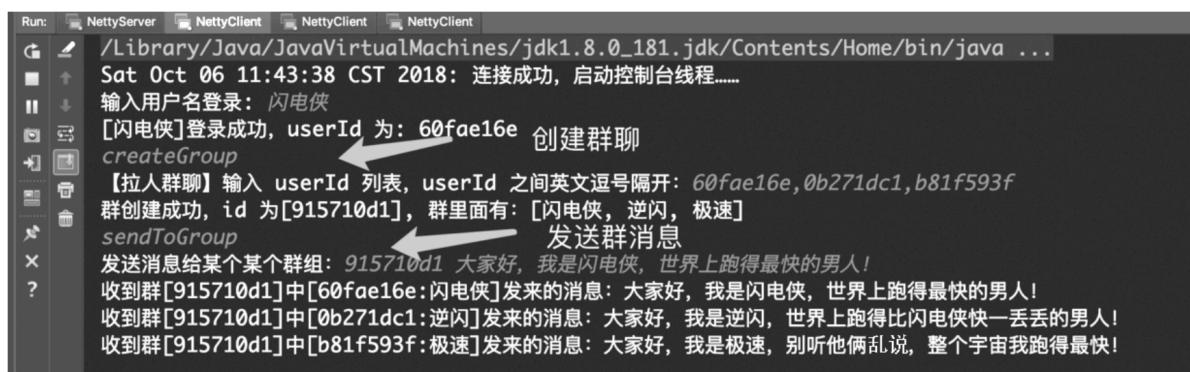
服务端



```
Run: NettyServer NettyClient NettyClient NettyClient
/Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/bin/java ...
Sat Oct 06 11:43:35 CST 2018: 端口[8000]绑定成功!
[闪电侠]登录成功
[逆闪]登录成功
[极速]登录成功
群创建成功, id 为 915710d1, 群里面有: [闪电侠, 逆闪, 极速]
```

闪电侠、逆闪、极速先后登录，然后闪电侠拉逆闪、极速和自己加入群聊。下面我们来看一下各个客户端的控制台界面。

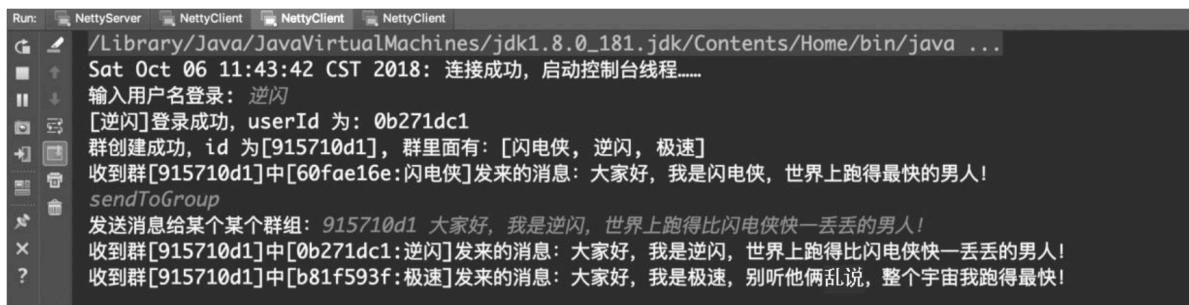
客户端（闪电侠）



```
Run: NettyServer NettyClient NettyClient NettyClient
/Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/bin/java ...
Sat Oct 06 11:43:38 CST 2018: 连接成功, 启动控制台线程.....
输入用户名登录: 闪电侠
[闪电侠]登录成功, userId 为: 60fae16e 创建群聊
createGroup
【拉人群聊】输入 userId 列表, userId 之间英文逗号隔开: 60fae16e,0b271dc1,b81f593f
群创建成功, id 为[915710d1], 群里面有: [闪电侠, 逆闪, 极速]
sendToGroup
发送群消息
发送消息给某个某个群组: 915710d1 大家好, 我是闪电侠, 世界上跑得最快的男人!
收到群[915710d1]中[60fae16e:闪电侠]发来的消息: 大家好, 我是闪电侠, 世界上跑得最快的男人!
收到群[915710d1]中[0b271dc1:逆闪]发来的消息: 大家好, 我是逆闪, 世界上跑得比闪电侠快一丢丢的男人!
收到群[915710d1]中[b81f593f:极速]发来的消息: 大家好, 我是极速, 别听他俩乱说, 整个宇宙我跑得最快!
```

闪电侠第一个输入sendToGroup发送群聊消息。

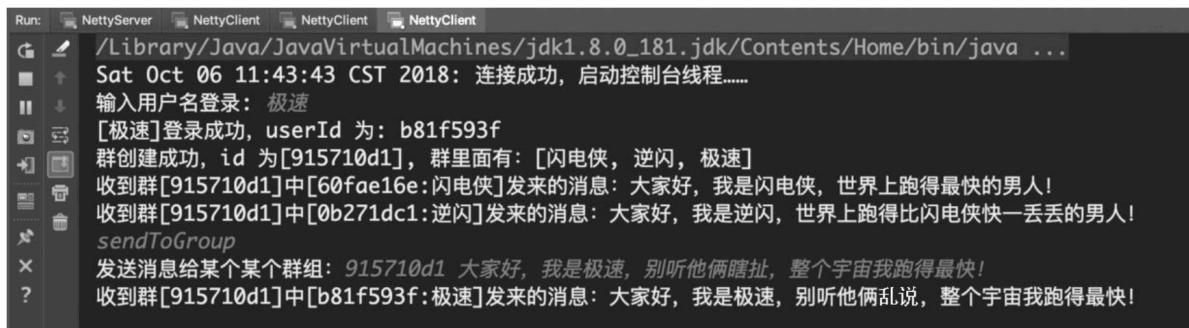
客户端（逆闪）



```
Run: NettyServer NettyClient NettyClient NettyClient
/Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/bin/java ...
Sat Oct 06 11:43:42 CST 2018: 连接成功, 启动控制台线程.....
输入用户名登录: 逆闪
[逆闪]登录成功, userId 为: 0b271dc1
群创建成功, id 为[915710d1], 群里面有: [闪电侠, 逆闪, 极速]
收到群[915710d1]中[60fae16e:闪电侠]发来的消息: 大家好, 我是闪电侠, 世界上跑得最快的男人!
sendToGroup
发送消息给某个某个群组: 915710d1 大家好, 我是逆闪, 世界上跑得比闪电侠快一丢丢的男人!
收到群[915710d1]中[0b271dc1:逆闪]发来的消息: 大家好, 我是逆闪, 世界上跑得比闪电侠快一丢丢的男人!
收到群[915710d1]中[b81f593f:极速]发来的消息: 大家好, 我是极速, 别听他俩乱说, 整个宇宙我跑得最快!
```

逆闪第二个输入sendToGroup发送群聊消息，他已经收到了闪电侠的消息。

客户端（极速）



```
Run: NettyServer NettyClient NettyClient NettyClient
/Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/bin/java ...
Sat Oct 06 11:43:43 CST 2018: 连接成功, 启动控制台线程.....
输入用户名登录: 极速
[极速]登录成功, userId 为: b81f593f
群创建成功, id 为[915710d1], 群里面有: [闪电侠, 逆闪, 极速]
收到群[915710d1]中[60fae16e:闪电侠]发来的消息: 大家好, 我是闪电侠, 世界上跑得最快的男人!
收到群[915710d1]中[0b271dc1:逆闪]发来的消息: 大家好, 我是逆闪, 世界上跑得比闪电侠快一丢丢的男人!
sendToGroup
发送消息给某个某个群组: 915710d1 大家好, 我是极速, 别听他俩瞎扯, 整个宇宙我跑得最快!
收到群[915710d1]中[b81f593f:极速]发来的消息: 大家好, 我是极速, 别听他俩乱说, 整个宇宙我跑得最快!
```

逆闪最后一个输入sendToGroup发送群聊消息，他已经收到了闪电侠和逆闪的消息。

- 1.在闪电侠的控制台，输入sendToGroup指令之后，再输入groupId+空格+消息内容，发送消息给群里的各位用户。随后，群里所有用户的控制台都显示了群聊消息。
- 2.陆续在逆闪和极速的控制台做相同的操作，群里所有用户的控制台陆续都显示了群聊消息。

这个实现过程和前面一样，下面我们仅关注核心实现部分。

19.2 群聊消息的收发实现

核心实现部分其实就是服务端处理群聊消息的Handler—GroupMessageRequestHandler。

GroupMessageRequestHandler.java

```
public class GroupMessageRequestHandler extends  
SimpleChannelInboundHandler <GroupMessageRequestPacket> {  
  
    @Override  
  
    protected void channelRead0(ChannelHandlerContext ctx,  
GroupMessageRequestPacket requestPacket) {  
  
        // 1.拿到 groupId 构造群聊消息的响应  
  
        String groupId = requestPacket.getToGroupId();  
  
        GroupMessageResponsePacket responsePacket = new  
GroupMessageResponsePacket();  
  
        responsePacket.setFromGroupId(groupId);  
  
        responsePacket.setMessage(requestPacket.getMessage());  
  
        responsePacket.setFromUser(SessionUtil.getSession(ctx.channel()));  
  
        // 2.拿到群聊对应的ChannelGroup，写到每个客户端  
  
        ChannelGroup channelGroup = SessionUtil.getChannelGroup(groupId);  
  
        channelGroup.writeAndFlush(responsePacket);  
  
    }  
  
}
```

1.通过groupId构造群聊消息的响应GroupMessageResponsePacket，然后把发送群聊的用户信息填入，我们直接复用与Channel绑定的Session作为这里的用户信息。

2.拿到群聊对应的ChannelGroup，通过writeAndFlush()方法写到客户端。

关于群聊功能实现相关的介绍到这里就差不多结束了，下面介绍本章的几个重要知识点，在生产环节中如何进行Netty性能优化。

19.3 共享Handler

在使用Netty完成一个即时聊天系统的核心功能之后，我们来仔细看一下服务端。

NettyServer.java

```
serverBootstrap  
  
.childHandler(new ChannelInitializer<NioSocketChannel>() {  
  
protected void initChannel(NioSocketChannel ch) {  
  
ch.pipeline().addLast(new Splitter());  
  
ch.pipeline().addLast(new PacketDecoder());  
  
ch.pipeline().addLast(new LoginRequestHandler());  
  
ch.pipeline().addLast(new AuthHandler());  
  
ch.pipeline().addLast(new MessageRequestHandler());  
  
ch.pipeline().addLast(new CreateGroupRequestHandler());  
  
ch.pipeline().addLast(new JoinGroupRequestHandler());  
  
ch.pipeline().addLast(new QuitGroupRequestHandler());  
  
ch.pipeline().addLast(new ListGroupMembersRequestHandler());  
  
ch.pipeline().addLast(new GroupMessageRequestHandler());  
  
ch.pipeline().addLast(new LogoutRequestHandler());  
  
ch.pipeline().addLast(new PacketEncoder());  
  
}  
  
} );
```

1.服务端的Pipeline链里已经有12个Handler，其中与指令相关的Handler有9个。

2.Netty在这里的逻辑是，每次有新连接到来的时候，都会调用ChannelInitializer的initChannel()方法，然后这9个与指令相关的Handler都会被创建一次。

3.可以看到，其实每一个指令Handler，它们内部都是没有成员变量的，也就是说是无状态的，我们完全可以让单例模式，即在调用pipeline().addLast()方法的时候，直接使用单例，不需要每次都创建，这样既提高了效率，也避免了创建很多小对象。

以LoginRequestHandler为例，我们来看一下如何改造。

LoginRequestHandler.java

```
// 1. 加上注解标识，表明该Handler可以是多个Channel共享的

@ChannelHandler.Sharable

public class LoginRequestHandler extends
SimpleChannelInboundHandler<LoginRequestPacket> {

    // 2. 构造单例

    public static final LoginRequestHandler INSTANCE = new
LoginRequestHandler();

    protected LoginRequestHandler() {
        }
    }
}
```

1.非常重要的是，如果一个Handler要被多个Channel共享，必须加上@ChannelHandler.Sharable显式地告诉Netty，这个Handler是支持多个Channel共享的，否则会报错。读者可以自行尝试一下。

2.仿照Netty源码里单例模式的写法，我们构造一个单例模式的类。

我们在服务端的代理里可以按照下面的格式来写。

NettyServer.java

```
serverBootstrap
```

```
.childHandler(new ChannelInitializer<NioSocketChannel>() {
    protected void initChannel(NioSocketChannel ch) {
        // 单例模式，多个Channel共享同一个Handler
        ch.pipeline().addLast(LoginRequestHandler.INSTANCE);
        // ...
    }
});
```

这样的话，每来一次新连接，添加Handler的时候就不需要每次都创建，对于剩下的8个指令，读者可以自行尝试改造一下。

19.4 压缩Handler——合并编解码器

当改造完之后，我们再来看一下服务端的代码。

NettyServer.java

```
serverBootstrap
    .childHandler(new ChannelInitializer<NioSocketChannel>() {
        protected void initChannel(NioSocketChannel ch) {
            ch.pipeline().addLast(new Splitter());
            ch.pipeline().addLast(new PacketDecoder());
            ch.pipeline().addLast(LoginRequestHandler.INSTANCE);
            ch.pipeline().addLast(AuthHandler.INSTANCE);
            ch.pipeline().addLast(MessageRequestHandler.INSTANCE);
            ch.pipeline().addLast(CreateGroupRequestHandler.INSTANCE);
            ch.pipeline().addLast(JoinGroupRequestHandler.INSTANCE);
        }
});
```

```

        ch.pipeline().addLast(QuitGroupRequestHandler.INSTANCE);

        ch.pipeline().addLast(ListGroupMembersRequestHandler.INSTANCE);

        ch.pipeline().addLast(GroupMessageRequestHandler.INSTANCE);

        ch.pipeline().addLast(LogoutRequestHandler.INSTANCE);

        ch.pipeline().addLast(new PacketEncoder());

    }

}

```

对于Pipeline中的第一个Handler—Spliter，我们是无法改动它的，因为它的内部实现与每个Channel都有关，每个Spliter都需要维持每个Channel当前读到的数据，也就是说它是有状态的。

而PacketDecoder与PacketEncoder是可以继续被改造的。Netty内部提供了一个类，叫作MessageToMessageCodec，使用它可以将编解码操作放到一个类中去实现。首先定义一个PacketCodecHandler。

PacketCodecHandler.java

```

@ChannelHandler.Sharable

public class PacketCodecHandler extends MessageToMessageCodec<ByteBuf,
Packet> {

    public static final PacketCodecHandler INSTANCE = new
PacketCodecHandler();

    private PacketCodecHandler() {
    }

    @Override

    protected void decode(ChannelHandlerContext ctx, ByteBuf byteBuf,
List<Object> out) {

```

```

        out.add(PacketCodec.INSTANCE.decode(byteBuf));

    }

@Override

protected void encode(ChannelHandlerContext ctx, Packet packet,
List<Object> out) {

    ByteBuf byteBuf = ctx.channel().alloc().ioBuffer();

    PacketCodec.INSTANCE.encode(byteBuf, packet);

    out.add(byteBuf);

}

}

```

1.这里的PacketCodecHandler是一个无状态的Handler，因此，同样可以使用单例模式来实现。

2.我们需要实现decode和encode方法，decode操作是将二进制数据ByteBuf转换为Java对象Packet；而encode操作则是一个相反的过程，在encode方法里，我们调用了Channel的内存分配器手工分配了ByteBuf。

接着，PacketDecoder和PacketEncoder都可以被删掉，服务端的代码就成了如下的样子。

```

serverBootstrap

.childHandler(new ChannelInitializer<NioSocketChannel>()  {

protected void initChannel(NioSocketChannel ch)  {

    ch.pipeline().addLast(new Splitter());

    ch.pipeline().addLast(PacketCodecHandler.INSTANCE);

    ch.pipeline().addLast(LoginRequestHandler.INSTANCE);

```

```
        ch.pipeline().addLast(AuthHandler.INSTANCE);

        ch.pipeline().addLast(MessageRequestHandler.INSTANCE);

        ch.pipeline().addLast(CreateGroupRequestHandler.INSTANCE);

        ch.pipeline().addLast(JoinGroupRequestHandler.INSTANCE);

        ch.pipeline().addLast(QuitGroupRequestHandler.INSTANCE);

        ch.pipeline().addLast(ListGroupMembersRequestHandler.INSTANCE);

        ch.pipeline().addLast(GroupMessageRequestHandler.INSTANCE);

        ch.pipeline().addLast(LogoutRequestHandler.INSTANCE);

    }

}

);
```

可以看到，除了拆包器，所有的Handler都写成了单例模式。当然，如果你的Handler里有与Channel相关的成员变量，那么就不要写成单例模式。不过，所有的状态其实都可以绑定在Channel的属性上，依然可以改造成单例模式。

这里有一个问题，为什么**PacketCodecHandler**这个Handler可以直接移到前面去，原来的**PacketEncoder**不是在最后吗？读者可以结合前面Handler与Pipeline相关的内容思考一下。

如果我们再仔细观察服务端的代码，就会发现，在Pipeline链中，绝大部分都是与指令相关的Handler。我们把这些Handler编排在一起，是为了逻辑简单，但是随着与指令相关的Handler越来越多，Handler链越来越长，事件传播过程中的性能损耗会被逐渐放大，因为解码器decode出来的每个Packet对象都要在每个Handler上经过一遍。我们接下来看一下如何缩短这个事件传播路径。

19.5 缩短事件传播路径

19.5.1 压缩Handler——合并平行Handler

对应用程序来说，每次decode出来一个指令对象后，其实都只会在一个指令Handler上进行处理。因此，可以把这么多的指令Handler压缩为一个Handler，我们来看一下如何实现。

首先定义一个IMHandler，实现如下。

IMHandler.java

```
@ChannelHandler.Sharable

public class IMHandler extends SimpleChannelInboundHandler<Packet> {

    public static final IMHandler INSTANCE = new IMHandler();

    private Map<Byte, SimpleChannelInboundHandler<? extends Packet>>
handlerMap;

    private IMHandler() {

        handlerMap = new HashMap<>();

        handlerMap.put(MESSAGE_REQUEST, MessageRequestHandler.INSTANCE);

        handlerMap.put(CREATE_GROUP_REQUEST,
CreateGroupRequestHandler.INSTANCE);

        handlerMap.put(JOIN_GROUP_REQUEST,
JoinGroupRequestHandler.INSTANCE);

        handlerMap.put(QUIT_GROUP_REQUEST,
QuitGroupRequestHandler.INSTANCE);

        handlerMap.put(LIST_GROUP_MEMBERS_REQUEST,
ListGroupMembersRequestHandler.INSTANCE);

        handlerMap.put(GROUP_MESSAGE_REQUEST,
GroupMessageRequestHandler.INSTANCE);

        handlerMap.put(LOGOUT_REQUEST, LogoutRequestHandler.INSTANCE);

    }

    @Override
```

```
protected void channelRead0(ChannelHandlerContext ctx, Packet
packet) throws
Exception {
    handlerMap.get(packet.getCommand()).channelRead(ctx, packet);
}
}
```

1. IMHandler是无状态的，依然可以写成一个单例模式的类。

2. 定义一个Map，存放指令到各个指令处理器的映射。

3. 每次回调IMHandler的channelRead0()方法的时候，都通过指令找到具体的Handler，然后调用指令Handler的channelRead()方法，其内部会进行指令类型转换，最终调用每个指令Handler的channelRead0()方法。

接下来，我们看一下如此压缩之后的服务端代码。

NettyServer.java

```
serverBootstrap
    .childHandler(new ChannelInitializer<NioSocketChannel>() {
        protected void initChannel(NioSocketChannel ch) {
            ch.pipeline().addLast(new Spliter());
            ch.pipeline().addLast(PacketCodecHandler.INSTANCE);
            ch.pipeline().addLast(LoginRequestHandler.INSTANCE);
            ch.pipeline().addLast(AuthHandler.INSTANCE);
            ch.pipeline().addLast(IMHandler.INSTANCE);
        }
    });
}
```

可以看到，现在服务端的代码已经变得很清爽了，所有的平行指令处理Handler都被压缩到了一个IMHandler中，并且IMHandler和指令Handler均为单例模式，在单机上有十几万甚至几十万连接的情况下，性能可以得到一定程度的提升，创建的对象也大大减少了。

当然，如果你对性能要求没这么高，大可不必搞得这么复杂，按照前面介绍的方式来实现即可。比如，我们的客户端在多数情况下是单连接的，其实并不需要搞得如此复杂，保持原样即可。

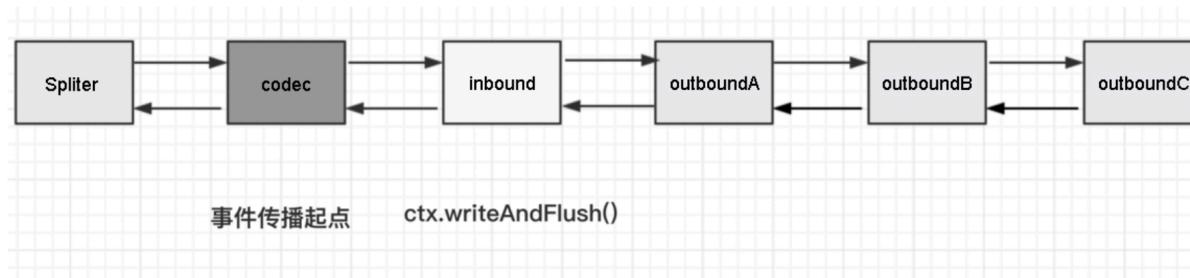
19.5.2 更改事件传播源

关于缩短事件传播路径，除了压缩Handler，还有一种方式就是，如果Outbound类型的Handler较多，在写数据的时候能用ctx.writeAndFlush()方法就用此方法。

ctx.writeAndFlush()方法的事件传播路径

ctx.writeAndFlush()方法从Pipeline链中的当前节点开始，往前找到第一个Outbound类型的Handler，把对象往前传播。如果这个对象确认不需要经过其他Outbound类型的Handler处理，那么就使用此方法。

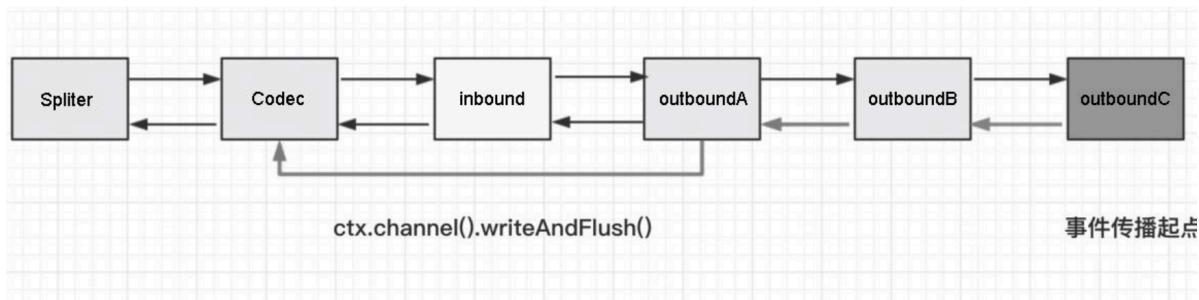
在某个Inbound类型的Handler处理完逻辑之后，调用ctx.writeAndFlush()方法可以直接一口气把对象送到codec中编码，然后写出去，如下图所示。



ctx.channel().writeAndFlush()方法的事件传播路径

ctx.channel().writeAndFlush()方法从Pipeline链中的最后一个Outbound类型的Handler开始，把对象往前传播。如果你确认当前创建的对象需要经过后面Outbound类型的Handler，那么就调用此方法。

在某个Inbound类型的Handler处理完逻辑之后，调用ctx.channel().writeAndFlush()方法，对象会从最后一个Outbound类型的Handler开始，逐个往前传播，路径要比ctx.writeAndFlush()方法长，如下图所示。



由此可见，在应用程序中，当我们没有改造编解码时，我们必须调用 `ctx.channel().writeAndFlush()` 方法，而经过改造之后，编码器（既属于 Inbound 又属于 Outbound 类型的 Handler）已处于 Pipeline 的最前面，因此，可以大胆使用 `ctx.writeAndFlush()` 方法。

19.6 减少阻塞主线程的操作

这部分内容可能稍微有些难度，如果不能理解，记住结论即可。

通常我们的应用程序会涉及数据库或者网络，比如下面这个例子。

```
protected void channelRead0(ChannelHandlerContext ctx, T packet) {
    // 1. 一些业务逻辑
    // 2. 数据库或者网络等一些耗时的操作
    // 3. writeAndFlush()
    // 4. 一些业务逻辑
}
```

我们看到，在 `channelRead0()` 方法的第二个过程中，有一些耗时的操作，这个时候，万万不能将这些操作直接就在这个方法中处理了，为什么？

在默认情况下，Netty 在启动的时候会开启两倍 CPU 核数个 NIO 线程，而通常情况下单机上会有几万或者十几万个连接，因此，一个 NIO 线程会管理着几千或几万个连接。在传播事件的过程中，单个 NIO 线程的处理逻辑可以抽象成以下步骤，下面以 `channelRead0()` 为例进行讲解。

单个NIO线程执行的抽象逻辑

```
List<Channel> channelList = 已有数据可读的Channel
for (Channel channel : channelList) {
    for (ChannelHandler handler : channel.pipeline()) {
        handler.channelRead0();
    }
}
```

从上面的抽象逻辑中可以看到，只要有一个Channel的一个Handler中的channelRead0()方法阻塞了NIO线程，最终都会拖慢绑定在该NIO线程上的其他所有Channel。当然，这里抽象的逻辑已经做了简化，想了解细节可以参考笔者关于Netty中NIO线程（即Reactor线程）文章的分析。

而我们需要怎么做？对于耗时的操作，我们需要把这些耗时的操作都丢到业务线程池中去处理，下面是解决方案的伪代码。

```
ThreadPool threadPool = xxx;

protected void channelRead0(ChannelHandlerContext ctx, T packet) {
    threadPool.submit(new Runnable() {
        // 1. 一些业务逻辑
        // 2. 数据库或者网络等一些耗时的操作
        // 3. writeAndFlush()
        // 4. 一些业务逻辑
    });
}
```

这样就可以避免一些耗时的操作影响Netty的NIO线程，从而影响其他Channel。

19.7 如何准确统计处理时长

我们接着前面的逻辑来讨论，通常应用程序都有统计某个操作响应时间的需求，比如，基于上面的例子我们会这么做。

```
protected void channelRead0(ChannelHandlerContext ctx, T packet) {  
  
    threadPool.submit(new Runnable() {  
  
        long begin = System.currentTimeMillis();  
  
        // 1. 一些业务逻辑  
  
        // 2. 数据库或者网络等一些耗时的操作  
  
        // 3. writeAndFlush()  
  
        // 4. 其他业务逻辑  
  
        long time = System.currentTimeMillis() - begin;  
  
    } );  
}
```

这种做法其实是不推荐的，为什么？因为writeAndFlush()方法如果在非NIO线程（这里，我们其实在业务线程中调用了该方法）中执行，它是一个异步的操作，调用之后，其实是会立即返回的，剩下的所有操作，都是由Netty内部的一个任务队列异步执行的。

因此，这里的writeAndFlush()方法执行完毕之后，并不能代表相关的逻辑，比如事件传播、编码等逻辑执行完毕，只是表示Netty接受了这个任务，那么如何才能判断writeAndFlush()方法执行完毕呢？我们可以这么做。

```
protected void channelRead0(ChannelHandlerContext ctx, T packet) {  
  
    threadPool.submit(new Runnable() {  
  
        long begin = System.currentTimeMillis();  
  
        // 1. 一些业务逻辑  
  
        // 2. 数据库或者网络等一些耗时的操作  
  
        // 3. writeAndFlush()  
  
        xxx.writeAndFlush().addListener(future -> {  
  
            if (future.isDone()) {  
                ...  
            }  
        });  
    } );  
}
```

```
// 4. 其他业务逻辑

long time = System.currentTimeMillis() - begin;

}

} );

} )
```

writeAndFlush()方法会返回一个ChannelFuture对象，我们给这个对象添加一个监听器，然后在回调方法里，可以监听这个方法执行的结果，进而执行其他逻辑，最后统计耗时，这样统计出来的耗时才是最准确的。

需要注意的是，Netty里很多方法都是异步的操作，在业务线程中如果要统计这部分操作的时间，都需要使用监听器回调的方式来统计耗时。如果在NIO线程中调用，则不需要这么做。

19.8 总结

本章的知识点较多，每一个知识点都是笔者在线上千万级长连接应用中摸索总结出来的实践经验，了解这些知识点会对你的线上应用有较大帮助。

1. 我们先在开头实现了群聊消息的最后一个部分：群聊消息的收发，这部分内容对大家来说已经非常平淡无奇了，因此没有贴完整的实现代码，重点在于实现了最后一步接下来所做的改造和优化。
2. 所有指令都实现之后，我们的Handler已经非常臃肿庞大了。接下来，我们通过单例模式改造、编解码器合并、平行指令Handler合并、慎重选择两种类型的writeAndFlush()方法等方式来压缩优化。
3. 在Handler的处理中，如果有耗时的操作，则需要把这些操作都丢到自定义的业务线程池中处理，因为NIO线程是有很多Channel共享的，我们不能阻塞它们。
4. 对于统计耗时的场景，如果在自定义业务线程中调用类似writeAndFlush()方法的异步操作，则需要通过添加监听器的方式来统计。

第20章

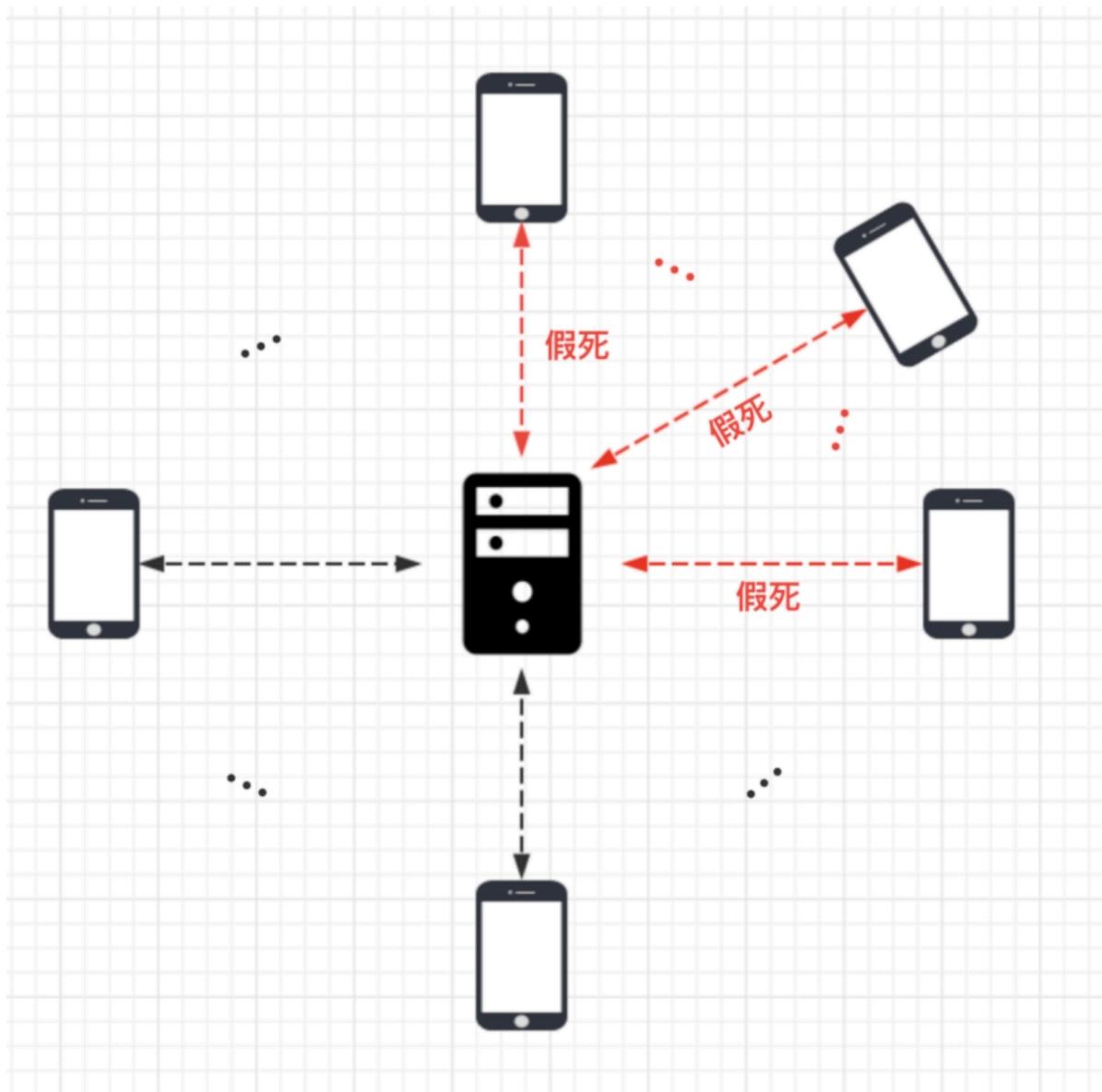
心跳与空闲检测

在本章中，我们一起探讨最后一个话题：心跳与空闲检测。

首先，我们来看一下客户端与服务端之间的网络会存在什么问题？

20.1 网络问题

下图是网络应用程序中普遍会遇到的一个问题：连接假死。



连接假死的现象是：在某一端（服务端或者客户端）看来，底层的TCP连接已经断开，但是应用程序并没有捕获到，因此会认为这条连接仍然是存在的。从TCP层面来说，只有收到四次握手数据包或者一个RST数据包，才表示连接的状态已断开。

连接假死会带来以下两大问题。

1.对于服务端来说，因为每个连接都会耗费CPU和内存资源，大量假死的连接会逐渐耗光服务器的资源，最终导致性能逐渐下降，程序崩溃。

2.对于客户端来说，连接假死会造成发送数据超时，影响用户体验。

通常，连接假死由以下几个原因造成。

1. 应用程序出现线程堵塞，无法进行数据的读写。
2. 客户端或者服务端网络相关的设备出现故障，比如网卡、机房故障。
3. 公网丢包。公网环境相对内网而言，非常容易出现丢包、网络抖动等现象，如果在一段时间内用户接入的网络连续出现丢包现象，那么对客户端来说，数据一直发送不出去；而服务端也一直收不到客户端的数据，连接就一直耗着。

如果应用程序是面向用户的，那么公网丢包这个问题出现的概率是非常高的。对于内网来说，内网丢包、抖动也会有一定概率发生。一旦出现此类问题，客户端和服务端都会受到影响。接下来，我们分别从服务端和客户端的角度来解决连接假死的问题。

20.2 服务端空闲检测

对于服务端来说，客户端的连接如果出现假死，那么服务端将无法收到客户端的数据。也就是说，如果能一直收到客户端发来的数据，则说明这个连接还是活的。因此，服务端对于连接假死的应对策略就是空闲检测。

何为空闲检测？空闲检测指的是每隔一段时间，检测这段时间内是否有数据读写。简化一下，服务端只需要检测一段时间内，是否收到过客户端发来的数据即可，Netty自带的IdleStateHandler就可以实现这个功能。

首先，我们写一个继承自IdleStateHandler的类，来定义检测到假死连接之后的逻辑。

IMIdleStateHandler.java

```
public class IMIdleStateHandler extends IdleStateHandler {  
  
    private static final int READER_IDLE_TIME = 15;  
  
    public IMIdleStateHandler() {  
  
        super(READER_IDLE_TIME, 0, 0, TimeUnit.SECONDS);  
  
    }  
  
    @Override protected void channelIdle(ChannelHandlerContext ctx,  
    IdleStateEvent evt) {  
  
        System.out.println(READER_IDLE_TIME + "秒内未读到数据，关闭连接");  
    }  
}
```

```
    ctx.channel().close();

}

}
```

1. 我们观察一下IMIdleStateHandler的构造函数，它调用父类IdleStateHandler的构造函数，有四个参数，其中第一个参数是读空闲时间，指的是在这段时间内如果没有读到数据，就表示连接假死；第二个参数是写空闲时间，指的是在这段时间如果没有写数据，就表示连接假死；第三个参数是读写空闲时间，指的是在这段时间内如果没有产生数据读或者写，就表示连接假死，写空闲和读写空闲均为0；最后一个参数是时间单位，在这个例子中表示的是：如果15秒内没有读到数据，就表示连接假死。

2. 连接假死之后会回调channelIdle()方法，我们在这个方法里打印消息，并手动关闭连接。

然后，我们把这个Handler插到服务端Pipeline的最前面。

NettyServer.java

```
serverBootstrap

    .childHandler(new ChannelInitializer<NioSocketChannel>()  {

        protected void initChannel(NioSocketChannel ch)  {

            // 空闲检测

            ch.pipeline().addLast(new IMIdleStateHandler());

            ch.pipeline().addLast(new Splitter());

            // ...

        }

    });

}
```

为什么要插到最前面？是因为假如插到最后面，如果这个连接读到了数据，但是在inbound传播的过程中出错了或者数据处理完毕就不往后传递了（我们的应用程序属于这类），那么最终IMIdleStateHandler就不会读到数据，会导致误判。

服务端的空闲检测完毕之后，我们再思考一下，在一段时间内没有读到客户端的数据，是否一定能判断连接假死呢？并不能，如果在这段时间内客户端确实没有发送数据过来，但是连接是正常的，那么这个时候服务端也不能关闭这个连接。为了防止服务端误判，我们还需要在客户端做点什么。

20.3 客户端定时发心跳数据包

服务端在一段时间内没有收到客户端的数据，这个现象产生的原因可以分为以下两种。

- 1.连接假死。
- 2.非假死状态下确实没有发送数据。

我们只需要排除第二种可能，那么连接自然就是假死的。要排查第二种情况，我们可以在客户端定期发送数据包到服务端，通常这个数据包被称为心跳数据包。我们定义一个Handler，定期发送心跳数据包给服务端。

HeartBeatTimerHandler.java

```
public class HeartBeatTimerHandler extends  
ChannelInboundHandlerAdapter {  
  
    private static final int HEARTBEAT_INTERVAL = 5;  
  
    @Override public void channelActive(ChannelHandlerContext ctx)  
throws Exception {  
  
    scheduleSendHeartBeat(ctx);  
  
    super.channelActive(ctx);  
  
}  
  
private void scheduleSendHeartBeat(ChannelHandlerContext ctx) {  
  
    ctx.executor().schedule(() -> {  
  
        if (ctx.channel().isActive()) {  
  
            ctx.writeAndFlush(new HeartBeatRequestPacket());  
        }  
    },  
    HEARTBEAT_INTERVAL,  
    TimeUnit.SECONDS);  
}
```

```
        scheduleSendHeartBeat(ctx);

    }

} , HEARTBEAT_INTERVAL, TimeUnit.SECONDS);

}

}
```

ctx.executor()方法返回的是当前Channel绑定的NIO线程。NIO线程有一个schedule()方法，类似JDK的延时任务机制，可以隔一段时间执行一个任务。这里实现了每隔5秒向服务端发送一个心跳数据包，这个间隔时间通常要比服务端的空闲检测时间的一半短一些，可以直接定义为空闲检测时间的三分之一，主要是为了排除公网偶发的秒级抖动。

在实际生产环境中，发送心跳数据包间隔时间和空闲检测时间可以略长一些，设置为几分钟级别，具体应用可以具体对待，没有强制规定。

上面我们其实解决了服务端的空闲检测问题，服务端这个时候能够在一定时间段内关掉假死的连接，释放连接的资源，但是对于客户端来说，我们也需要检测假死的连接。

20.4 服务端回复心跳与客户端空闲检测

客户端的空闲检测其实和服务端一样，依旧是在客户端Pipeline的最前面插入IMIdleStateHandler。

NettyClient.java

```
bootstrap

    .handler(new ChannelInitializer<SocketChannel>()  {

        public void initChannel(SocketChannel ch)  {

            // 空闲检测

            ch.pipeline().addLast(new IMIdleStateHandler());

            ch.pipeline().addLast(new Splitter());
        }
    });
}
```

```
// ...
```

为了排除因为服务端在非假死状态下确实没有发送数据的情况，服务端也要定期发送心跳数据包给客户端。

其实在前面我们已经实现了客户端向服务端定期发送心跳数据包，服务端这边只要在收到心跳数据包之后回复客户端，给客户端发送一个心跳响应包即可。如果在一段时间内客户端没有收到服务端发来的数据包，则可以判定这个连接为假死状态。

因此，服务端的Pipeline中需要再加上一个Handler—HeartBeatRequestHandler，由于这个Handler的处理是无须登录的，所以，我们将该Handler放置在AuthHandler前面。

NettyServer.java

```
serverBootstrap

    ch.pipeline().addLast(new IMIdleStateHandler());

    ch.pipeline().addLast(new Splitter());

    ch.pipeline().addLast(PacketCodecHandler.INSTANCE);

    ch.pipeline().addLast(LoginRequestHandler.INSTANCE);

    // 加在这里

    ch.pipeline().addLast(HeartBeatRequestHandler.INSTANCE);

    ch.pipeline().addLast(AuthHandler.INSTANCE);

    ch.pipeline().addLast(IMHandler.INSTANCE);

}

}

);
```

HeartBeatRequestHandler相应的实现如下。

```
@ChannelHandler.Sharable
```

```
public class HeartBeatRequestHandler extends  
SimpleChannelInboundHandler<HeartBeatRequestPacket> {  
  
    public static final HeartBeatRequestHandler INSTANCE = new  
HeartBeatRequestHandler();  
  
    private HeartBeatRequestHandler() {  
  
    }  
  
    @Override protected void channelRead0(ChannelHandlerContext ctx,  
HeartBeatRequestPacket requestPacket) {  
  
    ctx.writeAndFlush(new HeartBeatResponsePacket());  
  
    }  
  
}
```

实现非常简单，只是简单地回复一个HeartBeatResponsePacket数据包即可。客户端在检测到假死连接之后，断开连接，然后可以有一定的策略去重连、重新登录等，这里就不展开介绍了，留给读者自行实现。

关于心跳与空闲检测相关的内容就讲解到这里，原理理解清楚之后并不难实现。最后，我们来对本章内容做一下总结。

20.5 总结

- 1.首先讨论了连接假死相关的现象及产生的原因。
- 2.要处理连接假死问题，首先要实现客户端与服务端定期发送心跳数据包。在这里，其实服务端只需要对客户端的定时心跳数据包进行回复即可。
- 3.客户端与服务端如果都需要检测连接假死，那么直接在Pipeline的最前面插入一个自定义IdleStateHandler，在channelIdle()方法里自定义连接假死之后的逻辑即可。
- 4.通常空闲检测时间比发送心跳数据包的间隔时间的两倍要长一些，这也是为了排除偶发的公网抖动，防止误判。

20.6 思考

1. IMIdleStateHandler能否实现单例模式，为什么？
2. 如何实现客户端在断开连接之后自动重连并重新登录？

下篇

源码分析

◆

第21章 服务端启动流程解析 ◆

第22章 Reactor线程模型解析 ◆

第23章 客户端连接接入流程解析 ◆

第24章 解码原理解析 ◆

第25章 ChannelPipeline解析 ◆

第26章 writeAndFlush解析 ◆

第27章 本书总结

第21章

服务端启动流程解析

本章主要讲述的是Netty在服务端启动过程中是如何绑定端口、启动服务的。在启动服务的过程中，读者将会了解到Netty各大核心组件，本章暂时不会详细描述这些组件，而是简单介绍各大组件是如何协同工作、一起构建Netty核心的。

21.1 服务端启动示例

```
public final class SimpleServer {  
  
    public static void main(String [] args) {  
  
        EventLoopGroup bossGroup = new NioEventLoopGroup();  
  
        EventLoopGroup workerGroup = new NioEventLoopGroup();  
  
        try {  
  
            ServerBootstrap b = new ServerBootstrap();  
  
            b.group(bossGroup, workerGroup)  
  
                .channel(NioServerSocketChannel.class)  
  
                .handler(new SimpleServerHandler())  
  
                .childHandler(new ChannelInitializer<SocketChannel>() {  
  
                    @Override  
  
                    public void initChannel(SocketChannel ch) {  
  
                    }  
  
                } );  
  
            ChannelFuture f = b.bind(8888).sync();  
        }  
    }  
}
```

```
f.channel().closeFuture().sync();

} finally {

bossGroup.shutdownGracefully();

workerGroup.shutdownGracefully();

}

}

private static class SimpleServerHandler extends
ChannelInboundHandlerAdapter {

@Override

public void channelActive(ChannelHandlerContext ctx) {

System.out.println("channelActive");

}

@Override

public void channelRegistered(ChannelHandlerContext ctx) {

System.out.println("channelRegistered");

}

@Override

public void handlerAdded(ChannelHandlerContext ctx) {
```

```
System.out.println("handlerAdded");

    }

}

}
```

在本章中，我们写了一个比较完整的服务端启动例子，绑定在8888端口，使用NIO模式。我们再来看看每个方法的作用。

- 1.EventLoopGroup：服务端的线程模型外观类。从字面意思可以了解到，Netty的线程模型是事件驱动型的，也就是说，这个线程要做的事情就是不停地检测IO事件、处理IO事件、执行任务，不断重复这三个步骤。
- 2.ServerBootstrap：服务端的一个启动辅助类。通过给它设置一系列参数来绑定端口启动服务。
- 3..group(bossGroup,workerGroup)：设置服务端的线程模型。读者可以先想象一下：在一个工厂里，我们需要两种类型的人干活，一种是老板，一种是工人。老板负责从外面接活，把接到的活分配给工人。放到这里，bossGroup的作用就是不断地接收新的连接，将新连接交给workerGroup来处理。
- 4..channel(NioServerSocketChannel.class)：设置服务员的IO类型为NIO。Netty通过指定Channel的类型来指定IO类型。Channel在Netty里是一大核心概念，可以理解为，一个Channel就是一个连接或者一个服务端bind动作，后面会细讲。
- 5..handler(new SimpleServerHandler())：表示在服务端启动过程中，需要经过哪些流程。这里SimpleServerHandler最终的顶层接口为ChannelHandler，是Netty的一大核心概念，表示数据流经过的处理器，可以理解为流水线上的每一道关卡。
- 6..childHandler(new ChannelInitializer<SocketChannel>)...：使用过Netty的读者应该知道，这里的方法体主要用于设置一系列Handler来处理每个连接的数据，也就是上面所说的，老板接到一个活之后，告诉每个工人这个活的固定步骤。
- 7.ChannelFuture f =b.bind(8888).sync()：绑定端口同步等待。这里就是真正的启动过程了，绑定端口8888，等服务端启动完毕，才会进入下一行代码。
- 8.f.channel().closeFuture().sync()：等待服务端关闭端口绑定，这里的作用其实是让程序不会退出。

9.bossGroup.shutdownGracefully()和workerGroup.shutdownGracefully(): 关闭两组事件循环，关闭之后，main方法就结束了。

上述代码可以很轻松地在本地运行，最终控制台的输出如下。

```
handlerAdded
```

```
channelRegistered
```

```
channelActive
```

为什么控制台会按顺序输出这些字符，接下来我们就深入细节一探究竟。

21.2 服务端启动的核心步骤

我们通过以上示例代码来理一理服务端启动的基本流程。

在上面的示例代码中，我们是通过ServerBootstrap这个辅助类来实现服务端启动的，给这个启动类设置一些参数，然后通过它的外观接口来实现启动。重点落在下面这行代码上。

```
b.bind(8888).sync();
```

【提示】 我们刚开始看源码时，在对细节没那么清楚的情况下可以借助IDE的Debug功能，单步执行，以确定程序运行的入口。

我们跟进到bind()方法。

ServerBootstrap.java

```
public ChannelFuture bind(int inetPort) {  
  
    return bind(new InetSocketAddress(inetPort));  
  
}
```

通过端口号创建一个InetSocketAddress对象，然后继续调用重载方法bind()。

ServerBootstrap.java

```
public ChannelFuture bind(SocketAddress localAddress) {
```

```
validate();

if (localAddress == null) {

    throw new NullPointerException("localAddress");

}

return doBind(localAddress);

}
```

validate()验证服务启动需要的必要参数，然后调用doBind()。

ServerBootstrap.java

```
private ChannelFuture doBind(final SocketAddress localAddress) {

    //...

    final ChannelFuture regFuture = initAndRegister();

    //...

    final Channel channel = regFuture.channel();

    //...

    doBind0(regFuture, channel, localAddress, promise);

    //...

    return promise;

}
```

在这里，笔者减掉了细枝末节，专注于核心方法，分别为initAndRegister()和doBind0()。

从方法名我们已经可以略窥一二，init代表初始化，register代表注册，bind代表绑定端口。联系NIO开发的基本流程，可能是把某个东西初始化之后注册到Selector上，

最后bind像是在本地绑定端口号。带着这些猜测，我们深入分析下去。

我们先来看一下initAndRegister()方法。

AbstractBootstrap.java

```
final ChannelFuture initAndRegister() {  
  
    Channel channel = null;  
  
    //...  
  
    // 1. 创建服务端 Channel  
  
    channel = channelFactory.newChannel();  
  
    //...  
  
    // 2. 初始化服务端 Channel  
  
    init(channel);  
  
    //...  
  
    // 3. 注册服务端 Channel  
  
    ChannelFuture regFuture = config().group().register(channel);  
  
    //...  
  
    return regFuture;  
  
}
```

同样地，笔者略去了其他细节，专注于骨干代码。initAndRegister()中的3个主要方法与doBind0()方法一起组合成了服务端启动的4个过程。

1.创建服务端Channel。

2.初始化服务端Channel。

3.注册服务端Channel。

4.绑定服务端端口。

接下来，我们就详细分析这4个过程。

21.3 创建服务端Channel

我们首先需要了解一下Channel的定义，Netty官方对Channel的描述如下：

A nexus to a network socket or a component which is capable of I/O operations such as read,write,connect, and bind.

Channel可以理解为一个网络连接，或者具有IO操作能力（读、写、连接、绑定）的组件。

这里的Channel，由于是在服务启动的时候创建的，可以和Socket编程中的ServerSocket对应，也就是上述Netty官方定义中的IO组件的概念。

通过前面的分析，我们已经知道Channel是通过ChannelFactory创建出来的，ChannelFactory的接口很简单。

```
public interface ChannelFactory<T extends Channel> extends  
io.netty.bootstrap.ChannelFactory<T> {  
    T newChannel();  
}
```

通过一个方法，我们查看ChannelFactory被赋值的地方。

AbstractBootstrap.java

```
public B channelFactory(ChannelFactory<? extends C> channelFactory) {  
    // ...  
    this.channelFactory = channelFactory;  
    return (B) this;  
}
```

最终我们发现，在这个方法中，`ChannelFactory`被新建出来。

AbstractBootstrap.java

```
public B channel(Class<? extends C> channelClass) {  
  
    if (channelClass == null) {  
  
        throw new NullPointerException("channelClass");  
  
    }  
  
    return channelFactory(new ReflectiveChannelFactory<C>  
(channelClass));  
  
}
```

在本节的示例程序中，我们调用`channel(channelClass)`方法，将`NioServerSocketChannel`作为`ReflectiveChannelFactory`的构造方法的参数创建出一个`ReflectiveChannelFactory`。

然后回到本节最开始创建`Channel`的代码。

```
channelFactory.newChannel();
```

我们就可以了解到，最终调用了`ReflectiveChannelFactory.newChannel()`方法，继续跟进。

```
public class ReflectiveChannelFactory<T extends Channel> implements  
ChannelFactory<T>  
  
{  
  
    private final Class<? extends T> clazz;  
  
    public ReflectiveChannelFactory(Class<? extends T> clazz) {  
  
        //...  
    }
```

```
this.clazz = clazz;

}

@Override

public T newChannel() {

//...

return clazz.newInstance();

//...

}

}
```

看到clazz.newInstance(), 我们就明白了, 原来是通过反射的方式来创建一个对象, 而这个class就是我们在ServerBootstrap中传入的NioServerSocketChannel.class。

所以, 绕了一圈, 最终创建Channel相当于调用默认构造函数创建一个 NioServerSocketChannel对象。

【提示】这里提一下, 读源码细节, 有两种读的方式, 一种是回溯, 比如用到某个对象的时候可以逐层往上追溯, 一定会找到该对象最开始被创建的代码区块; 还有一种方式就是自顶向下, 逐层分析, 一般用在分析某个具体的方法, 剥丁解牛, 最后拼接出完整的流程。

21.3.1 创建JDK底层Channel

接下来我们分析NioServerSocketChannel的默认构造函数, 看一下创建 NioServerSocketChannel的细节。

NioServerSocketChannel.java

```
private static final SelectorProvider DEFAULT_SELECTOR_PROVIDER =

SelectorProvider.provider();

public NioServerSocketChannel() {
```

```
this(newSocket(DEFAULT_SELECTOR_PROVIDER));

}

private static ServerSocketChannel newSocket(SelectorProvider
provider) {
    //...
    return provider.openServerSocketChannel();
}
```

通过SelectorProvider.openServerSocketChannel()创建一个ServerSocketChannel对象，这个对象就是JDK领域的对象。

21.3.2 创建Channel配置类

接下来，Netty把上述对象继续传递到以下方法。

NioServerSocketChannel.java

```
public NioServerSocketChannel(ServerSocketChannel channel) {
    super(null, channel, SelectionKey.OP_ACCEPT);

    config = new NioServerSocketChannelConfig(this,
        javaChannel().socket());
}
```

这里第一行代码就跑到父类里了，第二行代码创建一个NioServerSocketChannelConfig，其顶层接口为ChannelConfig，Netty官方的描述如下。

A set of configuration properties of a Channel.

可以猜到，ChannelConfig也是Netty里的一个基本组件，初次看源码，看到这里，我们大可不必深挖这个对象，而是在用到的时候再回来深究。只要记住，这个对象在创建NioServerSocketChannel对象的时候被创建即可。

21.3.3 设置Channel类型为非阻塞

我们继续追踪NioServerSocketChannel的父类。

AbstractNioMessageChannel.java

```
protected AbstractNioMessageChannel(Channel parent, SelectableChannel
ch, int
readInterestOp)  {

    super(parent, ch, readInterestOp);

}
```

继续往上追。

AbstractNioChannel.java

```
protected AbstractNioChannel(Channel parent, SelectableChannel ch, int
readInterestOp)  {

    super(parent);

    this.ch = ch;

    this.readInterestOp = readInterestOp;

//...

    ch.configureBlocking(false);

//...

}
```

这里简单地将前面provider.openServerSocketChannel()创建出来的
ServerSocketChannel保存到成员变量ch，然后调用ch.configureBlocking(false)设置
该Channel为非阻塞模式，Java NIO开发的同学看到这里应该比较面熟了吧？

readInterestOp，即前面层层传入的SelectionKey.OP_ACCEPT，表示这个服务端Channel关心的是ACCEPT事件，即处理新连接的接入。

21.3.4 创建Channel核心组件

接下来我们重点分析super(parent)（这里的parent其实是null，由NioSocketChannel逐层调用的重载构造函数传入）。

AbstractChannel.java

```
protected AbstractChannel(Channel parent) {  
  
    this.parent = parent;  
  
    id = newId();  
  
    unsafe = newUnsafe();  
  
    pipeline = newChannelPipeline();  
  
}
```

在这个构造方法中，我们看到Netty创建了三大组件，分别赋值到成员变量。

AbstractChannel.java

第一个组件是ChannelId。

```
id = newId();  
  
protected ChannelId newId() {  
  
    return DefaultChannelId.newInstance();  
  
}
```

id是Netty中每条Channel的唯一标识，类似Snowflake算法，通过机器号、进程号、时间戳、随机数等方式生成。

第二个组件是Unsafe。

AbstractChannel.java

```
unsafe = newUnsafe();  
  
protected abstract AbstractUnsafe newUnsafe();
```

查看Unsafe的如下定义。

Unsafe operations that should never be called from user-code.These methods are only provided to implement the actual transport, and must be invoked from an I/O thread.

Unsafe是Netty的又一核心概念，后面我们会分析到，可以暂时忽略这个概念，只需要知道这里的newUnsafe方法是抽象方法，最终调用的是NioServerSocketChannel或者其父类中对应的方法即可。

最后一个组件是ChannelPipeline。

AbstractChannel.java

```
pipeline = newChannelPipeline();  
  
protected DefaultChannelPipeline newChannelPipeline() {  
  
    return new DefaultChannelPipeline(this);  
  
}
```

查看顶层接口ChannelPipeline的如下定义。

A list of ChannelHandlers which handles or intercepts inbound events and outbound operations of a Channel.

从该类的文档中可以看出，该接口也是Netty的一大核心组件，它包含了一个ChannelHandler链表，用于处理或者拦截Channel的Inbound事件和outbound操作。

到这里，我们总算把服务端Channel的创建流程梳理完毕了。将这些细节串起来的时候，我们顺带提取出Netty的几大基本组件。

- Channel。

- ChannelConfig。

- ChannelId。

- Unsafe。

- Pipeline。

- ChannelHandler。

21.3.5 创建服务端Channel小结

用户调用方法bind(port)第一步就是通过反射的方式创建一个NioServerSocketChannel（即服务端Channel）对象，并且在创建过程中创建了一系列核心组件。

初次看代码的时候，我们的目标是跟到调用了JDK API的那几行代码，在跟代码的过程中，我们遇到的其他概念可以先记下来，等代码跟完，我们就可以自顶向下，逐层分析，我们会在后面的章节中深入剖析这些组件。

21.4 初始化服务端Channel

服务端Channel创建完成之后，就进入了初始化Channel的流程。

ServerBootstrap.java

```
@Override  
  
void init(Channel channel) throws Exception {  
  
    // 1. 设置服务端Channel的Option与Attr  
  
    // 2. 设置客户端Channel的Option与Attr  
  
    // 3. 配置服务端启动逻辑  
  
}
```

ServerBootstrap的这个init方法有点长，这里就不展示出来了。初次看到这个方法，读者可能会觉得这个方法有点烦琐，无从下手，但是我们可以通过庖丁解牛、逐步拆解的方式来进行分析。下面是拆解步骤。

21.4.1 设置服务端Channel的Option与Attr

ServerBootstrap.java

```
final Map<ChannelOption<?>, Object> options = options0();

synchronized (options) {

    channel.config().setOptions(options);

}

final Map<AttributeKey<?>, Object> attrs = attrs0();

synchronized (attrs) {

    for (Entry<AttributeKey<?>, Object> e: attrs.entrySet()) {

        @SuppressWarnings("unchecked")

        AttributeKey<Object> key = (AttributeKey<Object>) e.getKey();

        channel.attr(key).set(e.getValue());

    }

}

关于options0()及attrs0()的作用，读者如果遗忘了的话，可以参考第4.3节。
```

通过上面的代码可以看到，这里先调用options0()及attrs0()，然后将得到的options和attrs注入ChannelConfig或者Channel，这里的options0()和attrs0()得到的对象都是我们在服务端启动流程的示例代码中自行设置的，这个分析过程较为简单，所以就交给读者了。

21.4.2 设置客户端Channel的Option与Attr

ServerBootstrap.java

```
final EventLoopGroup currentChildGroup = childGroup;
```

```

final ChannelHandler currentChildHandler = childHandler;

final Entry<ChannelOption<?>, Object> [] currentChildOptions;

final Entry<AttributeKey<?>, Object> [] currentChildAttrs;

synchronized (childOptions) {

    currentChildOptions =
        childOptions.entrySet().toArray(new OptionArray(childOptions.size()));

}

synchronized (childAttrs) {

    currentChildAttrs =
        childAttrs.entrySet().toArray(new AttrArray(childAttrs.size()));

}

```

这里，和上面类似，只不过保存的是新连接对应的Option和Attr，我们需要记住，自定义的这两个属性集合分别保存在两个成员变量中。

21.4.3 配置服务端启动逻辑

ServerBootstrap.java

```

p.addLast(new ChannelInitializer<Channel>() {
    @Override
    public void initChannel(Channel ch) throws Exception {
        // 添加用户自定义的Handler
        final ChannelPipeline pipeline = ch.pipeline();
        ChannelHandler handler = config.handler();
    }
});

```

```
if (handler != null) {  
  
    pipeline.addLast(handler);  
  
}  
  
// 添加一个特殊的Handler，用于接收新连接  
  
ch.eventLoop().execute(new Runnable() {  
  
    @Override  
  
    public void run() {  
  
        pipeline.addLast(new ServerBootstrapAcceptor(  
  
            currentChildGroup, currentChildHandler, currentChildOptions,  
  
            currentChildAttrs));  
  
    }  
  
});  
  
}  
};
```

到了最后一步—p.addLast()，用于定义服务端启动过程中需要执行哪些逻辑。在上述代码中，我们看到，Netty把服务端启动过程中需要执行的逻辑分为两块，一块是添加用户自定义的处理逻辑到服务端启动流程，另一块是添加一个特殊的处理逻辑。我们重点分析第二块逻辑。

从名字上就可以看出来，ServerBootstrapAcceptor是一个接入器，接受新请求，把新的请求传递给某个事件循环器。我们先不对这个类的原理做过多分析，在后面的章节中，我们还会再次遇到这个类。

21.4.4 初始化服务端Channel小结

我们发现，其实init也没有启动服务，只是初始化了一些基本配置和属性，以及在服务端的启动逻辑中加入了一个接入器，用来专门接收新连接。我们继续分析服务端

启动核心步骤的第三个步骤。

21.5 注册服务端Channel

这一步，我们先分析如下方法。

AbstractBootstrap.java

```
config().group().register(channel);
```

调用NioEventLoop中的register。

NioEventLoop.java

```
public ChannelFuture register(Channel channel) {  
    return register(new DefaultChannelPromise(channel, this));  
}  
  
public ChannelFuture register(final ChannelPromise promise) {  
    ObjectUtil.checkNotNull(promise, "promise");  
    promise.channel().unsafe().register(this, promise);  
    return promise;  
}
```

我们跟进去，进入下面的逻辑。

AbstractUnsafe

```
public final void register(EventLoop eventLoop, final ChannelPromise  
promise) {  
    // 绑定事件循环器  
    AbstractChannel.this.eventLoop = eventLoop;  
}
```

```
// 注册Selector  
  
register0(promise);  
  
}
```

这里依然只专注于重点代码，先将EventLoop事件循环器绑定到服务端Channel NioServerSocket-Channel上，然后调用register0()。

AbstractUnsafe

```
private void register0(ChannelPromise promise) {  
  
    boolean firstRegistration = neverRegistered;  
  
    // 1. 调用 JDK 底层注册 Selector  
  
    doRegister();  
  
    neverRegistered = false;  
  
    registered = true;  
  
    // 2. 回调handlerAdded事件  
  
    pipeline.invokeHandlerAddedIfNeeded();  
  
    safeSetSuccess(promise);  
  
    // 3. 传播channelRegistered事件  
  
    pipeline.fireChannelRegistered();  
  
    // 4. 其他逻辑  
  
}
```

register0()可以分为以下4个步骤来解析。

21.5.1 调用JDK底层注册Selector

AbstractNioChannel.java

```
protected void doRegister() throws Exception {
    // ...
    for (; ; ) {
        try {
            selectionKey = javaChannel().register(eventLoop().selector, 0,
this);
        }
        return;
    } catch (CancelledKeyException e) {
        // ...
    }
}

protected SelectableChannel javaChannel() {
    return ch;
}
```

在这个步骤中，我们可以看到关于JDK底层的操作：首先拿到在前面过程中创建的JDK底层的Channel，然后调用JDK的register()方法，将this也即NioServerSocketChannel对象当作attachment绑定到JDK的Selector上，这样后续从Selector拿到对应的事件之后，就可以把Netty领域的Channel拿出来。读者需要记下这个知识点，后面会用到。

21.5.2 回调handlerAdded事件

接着调用invokeHandlerAddedIfNeeded()，于是，控制台打印出来的第一行内容如下。

```
handlerAdded
```

关于具体是如何调用的，后面我们详细剖析Pipeline的时候再讲。

21.5.3 传播channelRegistered事件

调用pipeline.fireChannelRegistered()之后，控制台的显示如下。

```
handlerAdded
```

```
channelRegistered
```

同样，我们后面再详细分析与事件传播相关的逻辑。

21.5.4 其他逻辑

```
if (isActive()) {  
    if (firstRegistration) {  
        // 是这行代码触发的 active 事件吗  
        pipeline.fireChannelActive();  
    } else if (config().isAutoRead()) {  
        beginRead();  
    }  
}
```

读到这，你可能会想当然地以为，控制台上最后一行内容是由下面这行代码输出的。

```
pipeline.fireChannelActive();
```

我们不妨先看一下isActive()方法。

```
@Override
```

```
public boolean isActive() {
```

```
    return javaChannel().socket().isBound();  
  
}
```

从目前的流程来看，我们并没有将一个ServerSocket绑定到一个address上，所以 isActive()返回false，不会调用pipeline.fireChannelActive()方法，那么最后一行内容到底是谁输出的呢？其实，只要熟练运用IDE，要定位函数调用栈，就比较简单了。

下面是使用IntelliJ IDEA定位函数调用的具体方法。

1. 我们先在最终输出字符的这一行代码处打一个断点，也就是SimpleServerHandler的System.out.println("channelActive")这一行，然后Debug，执行到这一行，IntelliJ自动拉起了调用栈。我们唯一要做的，就是移动方向键，看到函数完整的调用链。

2. 如果你看到当前方法的方法体是Runnable的run方法，那么先去掉其他断点，然后在提交Runnable对象方法的地方打一个断点，重新Debug。

比如我们首次Debug发现调用栈中对应的卡点为pipeline.fireChannelActive()。

```
if (!wasActive && isActive()) { // 第二次在此打断点  
  
    invokeLater(new Runnable() {  
  
        @Override  
  
        public void run() {  
  
            pipeline.fireChannelActive(); // 在此行停下  
  
        }  
  
    } );  
  
}
```

我们想看最初的调用，就得跳出来，断点打到if (!wasActive && isActive())，因为Netty里很多任务执行都是异步线程调用的，如果我们要查看最先发起的方法调用，就必须查看Runnable被提交的地方，逐次递归下去，就能找到那行“消失的代码”。

最终，通过这种方式，找到了pipeline.fireChannelActive()发起调用的代码，刚好就是下面的服务端启动流程中最后一个步骤中的doBind0()方法。接下来我们就暂时挂

起active事件的传播的解析，进入服务端启动的最后一个过程。

21.5.5 注册服务端Channel小结

服务端启动在这一步做的事情，就是把前面创建的JDK的Channel注册到Selector，并且把Netty领域的Channel当作一个attachment绑定上去，同时回调handlerAdded和channelRegistered事件。

21.6 绑定服务端端口

AbstractBootstrap.java

```
private static void doBind0(
    final ChannelFuture regFuture, final Channel channel,
    final SocketAddress localAddress, final ChannelPromise promise) {
    channel.eventLoop().execute(new Runnable() {
        public void run() {
            // ...
            channel.bind(localAddress,
                promise).addListener(ChannelFutureListener.
                    CLOSE_ON_FAILURE);
        }
    });
}
```

我们发现，在调用doBind0(...)方法的时候，是通过包装一个Runnable进行异步化的，至于为什么这么做，我们在后续分析Netty的线程模型时会介绍。

接下来进入channel.bind()方法。

AbstractChannel.java

```
@Override

public ChannelFuture bind(SocketAddress localAddress) {
    return pipeline.bind(localAddress);
}
```

发现调用的是Pipeline的bind()方法。

DefaultChannelPipeline.java

```
public final ChannelFuture bind(SocketAddress localAddress) {
    return tail.bind(localAddress);
}
```

如果你对tail不是很了解，没关系，后面谈到事件传播机制时会提到。这里，你要想知道接下来代码的走向，唯一一个比较好的方式就是Debug单步进入。由于篇幅原因，这里不再详细展开。

最后会执行到如下代码。

HeadContext

```
@Override

public void bind(
    ChannelHandlerContext ctx, SocketAddress localAddress,
    ChannelPromise promise)

throws Exception {
    unsafe.bind(localAddress, promise);
}
```

这里的Unsafe就是前面提到的AbstractUnsafe，准确地说，是NioMessageUnsafe。

我们进入它的bind()方法。

```
@Override

public final void bind(final SocketAddress localAddress, final
ChannelPromise promise)

{

// ...

boolean wasActive = isActive();

// ...

// 1. 调用JDK底层绑定端口

doBind(localAddress);

// 2. 传播 channelActive 事件

if (! wasActive && isActive()) {

    invokeLater(new Runnable() {

        @Override

        public void run() {

            pipeline.fireChannelActive();

        }

    });

}

safeSetSuccess(promise);

}
```

这段代码也可以分为两个步骤：实际绑定端口和传播active事件。

21.6.1 调用JDK底层绑定端口

doBind()方法也很简单。

```
protected void doBind(SocketAddress localAddress) throws Exception {  
  
    if (PlatformDependent.javaVersion() >= 7) {  
  
        //noinspection Since15  
  
        javaChannel().bind(localAddress, config.getBacklog());  
  
    } else {  
  
        javaChannel().socket().bind(localAddress, config.getBacklog());  
  
    }  
  
}
```

在这里，Netty甚至区分了不同版本调用的JDK方法。最终，执行完这里的几行代码后，本地对应的端口就可以接收新连接了。

21.6.2 传播channelActive事件

绑定完端口后，还会调用pipeline.fireChannelActive()方法。我们继续跟进代码。

HeadContext

```
public void channelActive(ChannelHandlerContext ctx) throws Exception {  
  
    ctx.fireChannelActive();  
  
    readIfIsAutoRead();  
  
}
```

这里的ctx.fireChannelActive()继续往下传播active事件，最终就会传播到示例代码SimpleServerHandler中的channelActive方法。至此，控制台的所有输出都得到了合理的解释。

但是，我们注意到，传播完active事件之后，还有一行代码readIfIsAutoRead()。这行代码是做什么的呢？我们继续分析。

21.6.3 注册ACCEPT事件

进入readIfIsAutoRead方法。

HandlerContext

```
private void readIfIsAutoRead() {  
  
    if (channel.config().isAutoRead()) {  
  
        channel.read();  
  
    }  
  
}
```

我们先简单分析一下isAutoRead方法的返回值。

DefaultChannelConfig.java

```
private volatile int autoRead = 1;  
  
public boolean isAutoRead() {  
  
    return autoRead == 1;  
  
}
```

由此可见，isAutoRead方法默认返回true，于是进入以下方法。

AbstractChannel.java

```
public Channel read() {
```

```
        pipeline.read();

    return this;

}
```

这里的调用逻辑有点长，就不再赘述，读者可根据源码阅读技巧自行分析。最终，会调用如下代码块。

AbstractNioUnsafe.java

```
protected void doBeginRead() throws Exception {

    final SelectionKey selectionKey = this.selectionKey;

    if (!selectionKey.isValid()) {

        return;

    }

    readPending = true;

    final int interestOps = selectionKey.interestOps();

    if ((interestOps & readInterestOp) == 0) {

        selectionKey.interestOps(interestOps | readInterestOp);

    }

}
```

这里的this.selectionKey就是我们在前面register步骤返回的对象。我们在register的时候，注册ops的值是0，表示此时还不关注任何事件，只是建立绑定关系而已。

我们回忆一下注册过程。

AbstractNioChannel.java

```
selectionKey = javaChannel().register(eventLoop().selector, 0, this)
```

这里相当于把注册过的ops取了出来，通过了if条件，然后调用。

```
selectionKey.interestOps(interestOps | readInterestOp);
```

而这里的readInterestOp就是前面newChannel的时候传入的SelectionKey.OP_ACCEPT，所以，在这一部分代码中，Netty实际上想做的就是，告诉JDK的Selector，现在一切工作就绪，就差把ACCEPT事件注册到Selector上了。

21.6.4 绑定服务端端口小结

绑定服务端端口，最终会调用JDK绑定的API去进行实际绑定。绑定成功之后，会传播channelActive事件。最后，Netty会把ACCEPT事件注册到Selector上。这样，后续就可以通过Selector监测新连接的接入，从而做一些处理。

21.7 总结

最后，我们对本章内容进行总结，Netty启动一个服务端的流程如下。

1. 创建JDK底层Channel，创建对应Config对象，设置该Channel为非阻塞模式。
2. 创建Server对应的Channel，创建各大组件，包括ChannelConfig、ChannelId、ChannelPipeline、ChannelHandler、Unsafe等。
3. 初始化Server对应的Channel，设置Option、Attr，以及设置子Channel的Option、Attr，给Server的Channel添加连接接入器，用于接收新连接，并触发addHandler、register等事件。
4. 调用JDK底层注册Selector，将Netty领域的Channel当作attachment注册到Selector。
5. 调用JDK底层做端口绑定，并触发active事件。当active事件被触发时，才真正做服务端口绑定。

第22章

Reactor线程模型解析

Reactor线程模型是Netty高性能的关键要素之一。在第21章服务端启动的例子中，我们着重分析了服务端的启动流程。本章我们根据这个例子分析一下Netty的Reactor线程模型，也就是我们用得比较多的NioEventLoopGroup。

大家都知道，如果要基于JDK的API自己实现NIO编程，则需要一个线程池来不断监听端口。接收到新连接之后，这条连接上数据的读写会在另外一个线程池中进行。当然，这个线程池可以复用监听端口的线程池，不过一般不推荐。

在第21章中，bossGroup对应的就是监听端口的线程池，在绑定一个端口的情况下，这个线程池里只有一个线程；workerGroup对应的是连接的数据读写的线程。

那么Netty是如何创建NioEventLoopGroup也就是我们理解的线程池的？又是如何启动线程池里的线程的？线程启动又做了些什么事？这就是本章要分析的内容。

22.1 NioEventLoopGroup的创建

这部分，我们着重分析上一章Demo中的下面两行代码。

```
EventLoopGroup bossGroup = new NioEventLoopGroup();
EventLoopGroup workerGroup = new NioEventLoopGroup();
```

22.1.1 确定NioEventLoop的个数

在NioEventLoopGroup构造方法中，如果没有传递构造参数，那么默认构造参数为0，这个0在后面决定要创建多少个线程的时候会用上。

NioEventLoopGroup.java

```
public NioEventLoopGroup() {
    this(0);
}
```

```
public NioEventLoopGroup(int nThreads) {  
    this(nThreads, (Executor) null);  
}  
}
```

接下来就是构造方法的重载调用，这里省略中间过程，来到如下重载方法。

NioEventLoopGroup

```
public NioEventLoopGroup(int nThreads, Executor executor, final  
SelectorProvider  
  
selectorProvider,  
  
final SelectStrategyFactory selectStrategyFactory) {  
  
super(nThreads, executor, selectorProvider, selectStrategyFactory,  
  
RejectedExecutionHandlers.reject());  
}  
}
```

然后调用父类的构造方法。

MultithreadEventLoopGroup.java

```
protected MultithreadEventLoopGroup(int nThreads, Executor executor,  
Object... args) {  
  
super(nThreads == 0 ? DEFAULT_EVENT_LOOP_THREADS : nThreads,  
executor, args);  
}  
}
```

从这里其实可以分析出来，在Demo代码中，如果没有传递构造参数，那么这里的nThreads就是DEFAULT_EVENT_LOOP_THREADS，而如果传递了1，那么这里的nThreads就是1。

nThreads标识了最终线程池最多会创建多少个线程，默认最多会创建DEFAULT_EVENT_LOOP_THREADS个线程。

我们可以简单分析一下DEFAULT_EVENT_LOOP_THREADS是如何计算的。

MultithreadEventLoopGroup.java

```
private static final int DEFAULT_EVENT_LOOP_THREADS;

static {
    DEFAULT_EVENT_LOOP_THREADS = Math.max(1, SystemPropertyUtil.getInt(
        ("io.netty.eventLoopThreads",
        Runtime.getRuntime().availableProcessors() * 2)));
}
```

可以看到，如果没有设置程序启动参数，那么默认线程的个数为CPU核数乘以2，默认线程数为两倍CPU核数这个知识点在后面的内容中还会用到，读者需要重点关注一下。

22.1.2 NioEventLoopGroup的创建总体框架

我们已经知道，Netty是如何确定最终创建多少个线程的。接下来分析NioEventLoopGroup的线程创建过程。接着上一部分的调用过程，我们来到如下代码。

MultithreadEventExecutorGroup.java

```
protected MultithreadEventExecutorGroup(int nThreads, Executor
executor, Object...
args) {
    this(nThreads, executor,
DefaultEventExecutorChooserFactory.INSTANCE, args);
}
```

这里的第三个参数，我们重点关注一下，后续会重点分析。读者需要记下这里使用的对象实例是DefaultEventExecutorChooserFactory.INSTANCE。

这里只展示我们需要重点关注的代码，其他不太重要的细节均省略。

MultithreadEventExecutorGroup.java

```
protected MultithreadEventExecutorGroup(int nThreads, Executor
executor,
                                         EventExecutorChooserFactory chooserFactory, Object... args) {

    // 1. 创建 ThreadPerTaskExecutor

    if (executor == null) {

        executor = new ThreadPerTaskExecutor(newDefaultThreadFactory());

    }

    // 2. 创建 NioEventLoop

    children = new EventExecutor [nThreads] ;

    for (int i = 0; i < nThreads; i++) {

        children [i] = newChild(executor, args);

    }

    // 3. 创建线程选择器

    chooser = chooserFactory.newChooser(children);

}

}
```

重点就在于下面这3部分。

1.创建ThreadPerTaskExecutor: ThreadPerTaskExecutor表示每次调用execute()方法的时候，都会创建一个线程。

2.创建NioEventLoop: NioEventLoop对应线程池里线程的概念，这里其实就是用一个for循环创建的。

3.创建线程选择器：线程选择器的作用是确定每次如何从线程池中选择一个线程，也就是每次如何从NioEventLoopGroup中选择一个NioEventLoop。

下面我们将详细分析一下这3个过程。

22.1.3 创建ThreadPerTaskExecutor

MultithreadEventExecutorGroup.java

```
protected MultithreadEventExecutorGroup(int nThreads, Executor
executor,
                                         EventExecutorChooserFactory chooserFactory, Object... args) {
    // 1. 创建 ThreadPerTaskExecutor
    if (executor == null) {
        executor = new ThreadPerTaskExecutor(newDefaultThreadFactory());
    }
    // 2. 创建 NioEventLoop
    // 3. 创建线程选择器
}
```

其中ThreadPerTaskExecutor的代码不多，我们看一下它的execute方法。

ThreadPerTaskExecutor.java

```
private final ThreadFactory threadFactory;
public void execute(Runnable command) {
    threadFactory.newThread(command).start();
}
```

这个类的作用是，每次执行execute方法的时候，都会调用threadFactory来创建一个线程，把需要执行的命令传递进去，然后执行。

这里的threadFactory是通过以下方法创建的。

MultithreadEventExecutorGroup.java

```
protected ThreadFactory newDefaultThreadFactory() {  
  
    return new DefaultThreadFactory(getClass());  
  
}
```

这里的getClass()调用的对象是NioEventLoopGroup，因为我们是通过NioEventLoopGroup的构造方法层层调用到这里的。

接下来，我们分析一下DefaultThreadFactory，当读者了解了下面这段逻辑之后，就可以知道Netty关于线程的命令逻辑了。比如，为什么Netty的线程命名都类似“nioEventLoop-2-3”。

DefaultThreadFactory.java

```
// 这里的 poolType 是 NioEventLoopGroup  
  
public DefaultThreadFactory(Class<?> poolType) {  
  
    this(poolType, false, Thread.NORM_PRIORITY);  
  
}  
  
// 这里的toPoolName方法的最终结果就是把NioEventLoopGroup的首字母变成小写，也就是
```

```
NioEventLoopGroup
```

```
public DefaultThreadFactory(Class<?> poolType, boolean daemon, int priority) {  
    this(toPoolName(poolType), daemon, priority);  
}
```

接下来，会调用：

DefaultThreadFactory.java

```
private static final AtomicInteger poolId = new AtomicInteger();  
  
private final String prefix;  
  
public DefaultThreadFactory(String poolName, boolean daemon, int priority, ThreadGroup  
threadGroup) {  
  
    prefix = poolName + '-' + poolId.incrementAndGet() + '-';  
}
```

省去其他无关信息，我们看到，最终DefaultThreadFactory会用一个成员变量prefix来标记线程名的前缀，其中poolId是全局的自增ID。读者也不难分析，prefix的最终格式为nioEventLoopGroup-线程池编号-。

了解了上面的信息后，我们来分析一下DefaultThreadFactory的新线程方法。

DefaultThreadFactory.java

```
private final AtomicInteger nextId = new AtomicInteger();  
  
public Thread newThread(Runnable r) {  
    Thread t = newThread(new DefaultRunnableDecorator(r), prefix +  
nextId.incrementAndGet());
```

```

    return t;

}

protected Thread newThread(Runnable r, String name) {
    return new FastThreadLocalThread(threadGroup, r, name);
}

```

可以看到，最终创建出来的线程名是prefix加一个自增的nextId。这里的nextId是对象级别的成员变量，只在一个NioEventLoopGroup里递增。所以，我们最终看到，Netty里的线程名字都类似于NioEventLoopGroup-2-3，表示这个线程是属于第几个NioEventLoopGroup的第几个NioEventLoop。

我们还注意到，DefaultThreadFactory创建出来的线程实体是经过Netty优化之后的FastThreadLocalThread，也可以理解为，这个类型的线程实体在操作ThreadLocal的时候，要比JDK快，这部分内容的分析，我们放到后续章节中。

到这里，我们已经了解到，Netty的线程实体是由ThreadPerTaskExecutor创建的，ThreadPerTaskExecutor每次执行execute的时候都会创建一个FastThreadLocalThread的线程实体。接下来，我们就分析一下NioEventLoopGroup创建总体框架的第二个过程。

22.1.4 创建NioEventLoop

MultithreadEventExecutorGroup.java

```

protected MultithreadEventExecutorGroup(int nThreads, Executor
executor,
                                         EventExecutorChooserFactory chooserFactory, Object... args) {
    // 1. 创建 ThreadPerTaskExecutor
    // 2. 创建 NioEventLoop
    children = new EventExecutor [nThreads] ;
    for (int i = 0; i < nThreads; i++) {

```

```
children [i] = newChild(executor, args);

}

// 3. 创建线程选择器

}
```

Netty使用for循环来创建nThreads个NioEventLoop，通过前面的分析，我们可能已经猜到，一个NioEventLoop对应一个线程实体，这个线程实体是FastThreadLocalThread。

我们先来分析newChild方法。

NioEventLoopGroup.java

```
protected EventLoop newChild(Executor executor, Object... args) throws
Exception {
    return new NioEventLoop(this, executor, (SelectorProvider) args [0] ,
        ((SelectStrategyFactory) args [1] ).newSelectStrategy(),
        (RejectedExecutionHandler)
    args [2] );
}

}
```

newChild传递一个executor参数，这个参数就是前面分析的ThreadPerTaskExecutor，而args参数是我们通过层层调用传递过来的一系列参数。

我们看到，newChild方法最终创建的是一个NioEventLoop对象，这里的this指的是NioEventLoopGroup，表示归属于哪个NioEventLoopGroup。

我们继续分析NioEventLoop的创建过程。

NioEventLoop.java

```
Selector selector;
```

```
NioEventLoop(NioEventLoopGroup parent, Executor executor,  
SelectorProvider  
  
selectorProvider,  
  
SelectStrategy strategy, RejectedExecutionHandler  
rejectedExecutionHandler) {  
  
super(parent, executor, false, DEFAULT_MAX_PENDING_TASKS,  
rejectedExecutionHandler);  
  
// ...  
  
selector = openSelector();  
  
}
```

我们同样略去了非关键代码。首先，继续调用父类构造方法；然后，通过调用openSelector方法来创建一个Selector。Selector是NIO编程里最核心的概念，一个Selector可以将多个连接绑定在一起，负责监听这些连接的读写事件，即多路复用。

在openSelector方法中，Netty通过反射对Selector底层的数据结构进行了优化。

我们继续分析NioEventLoop，往上层层调用父类构造方法，最终来到以下逻辑。

SingleThreadEventExecutor.java

```
private final Queue<Runnable> taskQueue;  
  
protected SingleThreadEventExecutor(EventExecutorGroup parent,  
Executor executor,  
  
boolean addTaskWakesUp, int maxPendingTasks,  
  
RejectedExecutionHandler rejectedHandler) {  
  
taskQueue = newTaskQueue(this.maxPendingTasks);  
  
}
```

这里最关键的代码其实只有一行，其作用是创建一个任务队列，Netty中所有的异步执行，本质上都是通过这个任务队列来协调完成的。

这里的newTaskQueue是一个protected方法，在NioEventLoop中被重写。

NioEventLoop.java

```
@Override  
  
protected Queue<Runnable> newTaskQueue(int maxPendingTasks) {  
  
    return PlatformDependent.newMpscQueue(maxPendingTasks);  
  
}
```

这里创建的是一个高性能的MPSC队列，也就是多生产者单消费者队列，单消费者指某个NioEventLoop对应的线程，而多生产者就是此NioEventLoop对应的线程之外的线程，通常情况下就是我们的业务线程。比如，我们在调用writeAndFlush的时候，可以不用考虑线程安全，随意调用，这些线程指的就是多消费者，在NioEventLoop的执行部分，我们会详细分析。

如果我们继续往下跟踪，会发现Netty的MPSC队列直接使用的JCTools，可以说Netty的高性能，很大程度上功劳要归功于这个工具包，感兴趣的读者可以了解一下。

关于NioEventLoop的创建，最关键的就是两部分：创建一个Selector和创建一个MPSC队列，这三者均为一对关系。

22.1.5 创建线程选择器

关于NioEventLoopGroup的最后一部分内容，就是创建线程选择器，那么线程选择器的作用是什么？

在传统的NIO编程中，一个新连接被创建后，通常需要给这个连接绑定一个Selector，之后这个连接的整个生命周期都由这个Selector管理。而从上面的代码中，我们分析到，Netty中一个Selector对应一个NioEventLoop，线程选择器的作用正是为一个连接在一个EventLoopGroup中选择一个NioEventLoop，从而将这个连接绑定到某个Selector上。

下面是线程选择器的接口描述。

EventExecutorChooserFactory.java

```
interface EventExecutorChooser {  
    /**  
     * Returns the new {@link EventExecutor} to use.  
     */  
  
    EventExecutor next();  
}
```

接下来分析NioEventLoopGroup创建总体框架的最后一个过程—创建线程选择器。

MultithreadEventExecutorGroup.java

```
protected MultithreadEventExecutorGroup(int nThreads, Executor  
executor,  
  
EventExecutorChooserFactory chooserFactory,  
  
Object... args) {  
  
    // 1. 创建 ThreadPerTaskExecutor  
  
    // 2. 创建 NioEventLoop  
  
    // 3. 创建线程选择器  
  
    chooser = chooserFactory.newChooser(children);  
  
}
```

在第22.1.2节中，我们知道这里的chooserFactory是DefaultEventExecutorChooserFactory，我们分析这个类的newChooser方法。

DefaultEventExecutorChooserFactory.java

```
public EventExecutorChooser newChooser(EventExecutor [] executors) {  
  
    if (isPowerOfTwo(executors.length)) {
```

```
        return new PowerOfTowEventExecutorChooser(executors);

    } else {

        return new GenericEventExecutorChooser(executors);

    }

}
```

比较有意思的是，Netty通过判断NioEventLoopGroup中的NioEventLoop是否是2的幂来创建不同的线程选择器，不管是哪一种选择器，最终效果都是从第一个NioEventLoop遍历到最后一个NioEventLoop，再从第一个开始，如此循环。

GenericEventExecutorChooser通过简单的累加取模来实现循环的逻辑。

GenericEventExecutorChooser.java

```
private final AtomicInteger idx = new AtomicInteger();

private final EventExecutor[] executors;

@Override

public EventExecutor next() {

    return executors [Math.abs(idx.getAndIncrement() %
executors.length)] ;

}
```

而PowerOfTowEventExecutorChooser是通过位运算实现的。

PowerOfTowEventExecutorChooser.java

```
private final AtomicInteger idx = new AtomicInteger();

private final EventExecutor[] executors;

@Override

public EventExecutor next() {
```

```
// 这里先计算executors.length - 1, 再进行与运算

return executors [idx.getAndIncrement() & executors.length - 1] ;

}
```

比如，如果CPU核数为4，默认创建8个NioEventLoop，符合2的幂，这里executors.length-1计算出来是7，也就是二进制数的111，那么，只要使用一个自增的ID，和111进行与运算，就能实现循环的效果，而与运算是操作系统底层支持的，要比取模运算效率高很多。由此，我们也可以感受到，为了性能优化，Netty在细节上面考虑得确实非常细致。

22.1.6 NioEventLoopGroup的创建小结

关于NioEventLoopGroup的创建，核心知识已经梳理完，我们来稍作总结。

- 1.在默认情况下，NioEventLoopGroup会创建两倍CPU核数个NioEventLoop，一个NioEventLoop和一个Selector及一个MPSC任务队列一一对应。
- 2.NioEventLoop线程的命名规则是nioEventLoopGroup-xx-yy，xx表示全局第xx个NioEventLoopGroup，yy表示这个NioEventLoop在NioEventLoopGroup中是第yy个。
- 3.线程选择器的作用是为一个连接选择一个NioEventLoop，如果NioEventLoop的个数为2的幂，则Netty会使用与运算进行优化。

22.2 NioEventLoop对应线程的创建和启动

22.2.1 NioEventLoop的启动入口

在前面的Demo中，我们在main方法中创建了两个NioEventLoopGroup：bossGroup和workerGroup。在服务端启动章节的分析中，注册服务端Channel到Selector的过程中会经过以下逻辑。

AbstractUnsafe

```
public final void register(EventLoop eventLoop, final ChannelPromise promise) {

    AbstractChannel.this.eventLoop = eventLoop;

    if (eventLoop.inEventLoop()) {
```

```
register0(promise);  
}  
else {  
    eventLoop.execute(new Runnable() {  
  
        @Override  
        public void run() {  
            register0(promise);  
        }  
    } );  
}  
}
```

这里只展示核心代码部分，当代码调用到这里的时候，当前线程是main方法对应的主线程，读者可以自行在这个地方打断点，然后根据我们在源码分析技巧中的查看调用栈来追溯整个调用过程。

22.2.2 创建线程并启动

我们了解到当前线程是什么之后，接下来分析一下inEventLoop这个方法。

AbstractEventExecutor.java

```
@Override  
  
public boolean inEventLoop()  {  
  
    return inEventLoop(Thread.currentThread());  
  
}  
  
@Override  
  
public boolean inEventLoop(Thread thread) {
```

```
    return thread == this.thread;

}
```

首先调用重载方法，将当前线程，也就是main方法对应的主线程传递进来，然后将这个线程与this.thread进行比较。由于this.thread此时并未赋值，所以为空，因而返回false。

另外，我们在Netty的源码中，会在很多地方看到inEventLoop这样的判断，这个方法的本质含义就是判断当前线程是否是Netty的Reactor线程，也就是NioEventLoop对应的线程实体。后面我们会看到，创建一个线程之后，会将这个线程实体保存到thread这个成员变量中。

因为这个地方返回false，所以接下来调用eventLoop.execute()这个逻辑。

SingleThreadEventExecutor.java

```
@Override

public void execute(Runnable task) {

    boolean inEventLoop = inEventLoop();

    if (inEventLoop) {

        addTask(task);

    } else {

        startThread();

        addTask(task);

    }
}
```

由于execute方法是public，因此可能被用户代码使用。比如，我们经常使用ctx.executor().execute(...)，所以，这里又进行了一次外部线程判断逻辑，确保执行task不会遇到线程安全问题。

这里是main方法对应的线程，所以接下来执行startThread方法。

SingleThreadEventExecutor.java

```
private void startThread() {  
  
    if (STATE_UPDATER.get(this) == ST_NOT_STARTED) {  
  
        if (STATE_UPDATER.compareAndSet(this, ST_NOT_STARTED, ST_STARTED)) {  
  
            doStartThread();  
  
        }  
  
    }  
  
}
```

可以看到，Netty会判断Reactor线程有没有被启动。如果没有被启动，则调用doStartThread方法启动线程。

SingleThreadEventExecutor.java

```
private void doStartThread() {  
  
    ...  
  
    executor.execute(new Runnable() {  
  
        @Override  
  
        public void run() {  
  
            thread = Thread.currentThread();  
  
            ...  
  
            SingleThreadEventExecutor.this.run();  
        }  
    });  
}
```

```
...
    }
}
}
```

在执行doStartThread的时候，会调用内部成员变量executor的execute方法，而根据我们在第22.1.3节的分析，executor就是ThreadPerTaskExecutor，这个对象的作用就是每次执行Runnable的时候，都会先创建一个线程再执行。

在这个Runnable中，通过一个成员变量thread来保存ThreadPerTaskExecutor创建出来的线程，这个线程就是我们在第22.1.3节中分析的FastThreadLocalThread。至此，我们终于知道，一个NioEventLoop是如何与一个线程实体绑定的：NioEventLoop通过ThreadPerTaskExecutor创建一个FastThreadLocalThread，然后通过一个成员变量来指向这个线程。

NioEventLoop保存完线程的引用之后，随即调用run方法。这个run方法就是Netty Reactor的核心所在。

通过这一部分的学习，我们基本理清了Netty在服务端启动过程中，boss对应的NioEventLoopGroup是如何创建第一个NioEventLoop线程的，又是如何开始线程的Reactor循环的，而在后面的客户端新连接接入过程中，我们会继续分析worker对应的NioEventLoop又是如何创建线程并启动的。

22.3 NioEventLoop的执行流程

22.3.1 NioEventLoop的执行总体框架

我们继续剖析NioEventLoop的执行流程。

NioEventLoop.java

```
@Override
protected void run() {
    for (; ; ) {
        // 1. 执行一次事件轮询
    }
}
```

```
select(wakenUp.getAndSet(false));

if (wakenUp.get()) {
    selector.wakeup();
}

// 2. 处理产生 IO 事件的 Channel

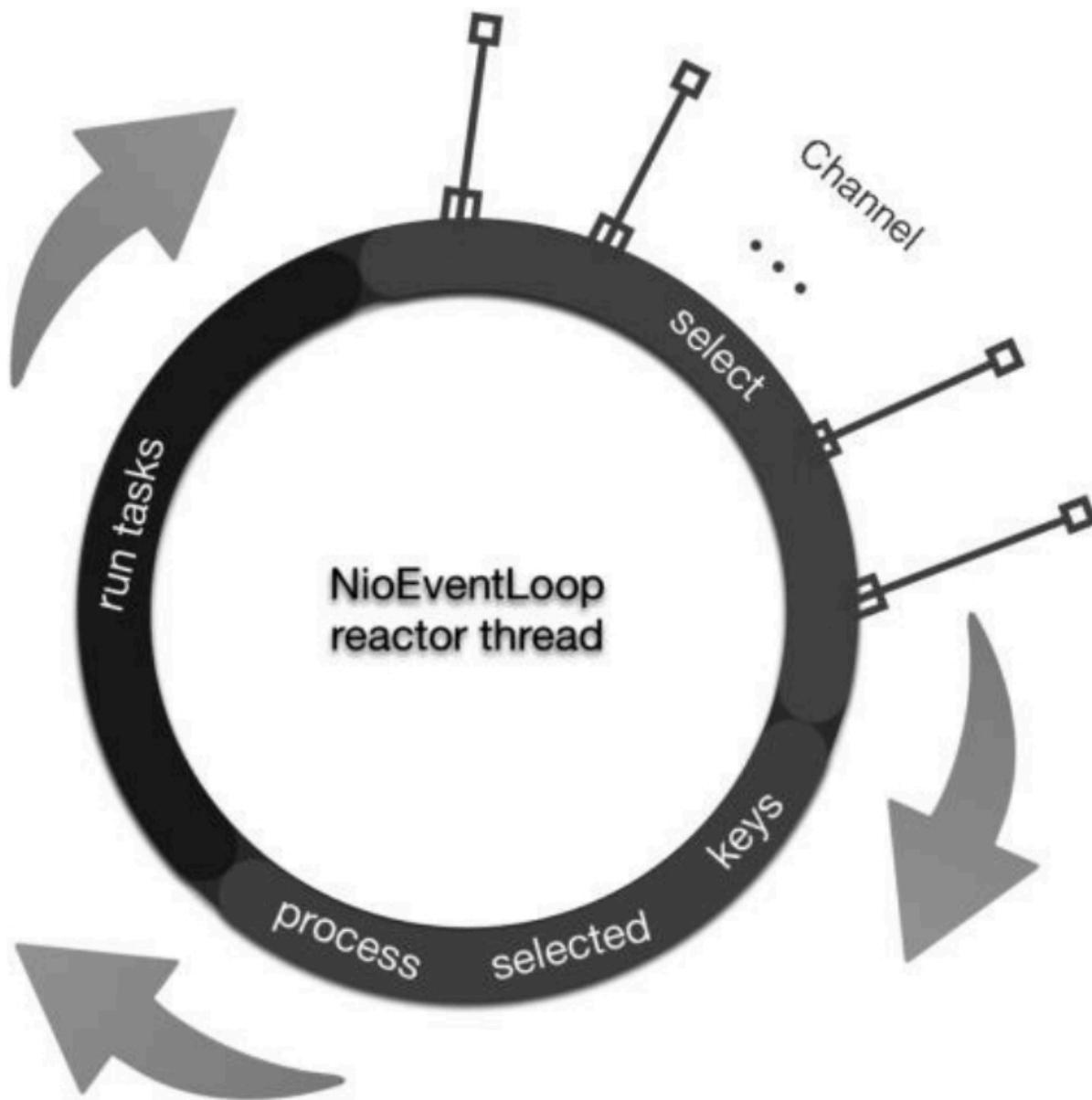
processSelectedKeys();

// 3. 执行任务

runAllTasks(...);

}
}
```

我们抽取出主干，发现Reactor线程做的事情其实很简单，用下图就可以说明。



Reactor线程大概做的事情为不断循环以下3个步骤。

1. 执行一次事件轮询。首先轮询注册到Reactor线程对应的Selector上的所有Channel的IO事件。

```
select(wakenUp.getAndSet(false));  
  
if (wakenUp.get()) {  
  
    selector.wakeup();  
  
}
```

2. 处理产生IO事件的Channel。如果有读写或者新连接接入事件，则处理：

```
processSelectedKeys();
```

3. 处理任务队列。

```
runAllTasks(...);
```

接下来，我们详细分析一下这3个步骤。

22.3.2 执行一次事件轮询

```
select(wakenUp.getAndSet(false));  
  
if (wakenUp.get()) {  
  
    selector.wakeup();  
  
}
```

wakenUp表示是否应该唤醒正在阻塞的select操作，可以看到Netty在进行一次新的循环逻辑之前，都会将wakenUp设置成false，标志新一轮循环的开始。具体的select操作我们也拆分开来看。

NioEventLoop.java

```
private void select(boolean oldWakenUp) throws IOException {  
  
    for(jj) {  
  
        // 1. 定时任务截止时间快到了，中断本次轮询  
  
        // 2. 轮询过程中发现有任务加入，中断本次轮询  
  
        // 3. 阻塞式 select 操作  
  
        // 4. 解决 JDK 的 NIO Bug  
  
    }  
  
}
```

相信大多数读者都或多或少听说过Netty解决了JDK“臭名昭著”的Epoll空轮询Bug，在本节中，读者将会了解到Netty是如何解决的。

接下来，我们详细分析一下Netty关于事件轮询的4段主要逻辑。

1.定时任务截止时间快到了，中断本次轮询。

NioEventLoop.java

```
int selectCnt = 0;

long currentTimeNanos = System.nanoTime();

long selectDeadLineNanos = currentTimeNanos +
delayNanos(currentTimeNanos);

for ( ; ) {

    // 1.定时任务截止时间快到了，中断本次轮询

    long timeoutMillis = (selectDeadLineNanos - currentTimeNanos +
500000L) / 1000000L;

    if (timeoutMillis <= 0) {

        if (selectCnt == 0) {

            selector.selectNow();

            selectCnt = 1;

        }

        break;

    }

    // 2.轮询过程中发现有任务加入，中断本次轮询

    // 3.阻塞式select操作
```

```
// 4.解决JDK的NIO Bug
```

```
}
```

我们可以看到， NioEventLoop中Reactor线程的select操作也是一个for循环。在for循环第一步中，如果发现当前的定时任务队列中有任务的截止时间快到了($<=0.5\text{ms}$)，就跳出循环。此外，跳出之前，如果发现目前为止还没有进行过select操作 (if (`selectCnt == 0`))，那么就调用一次`selectNow()`，该方法会立即返回，不会阻塞。

这里说明一点，Netty里的定时任务队列是按照延迟时间从小到大进行排序的，`delayNanos (currentTimeNanos)`方法即取出第一个定时任务的延迟时间。

SingleThreadEventExecutor.java

```
protected long delayNanos(long currentTimeNanos) {  
  
    ScheduledFutureTask<?> scheduledTask = peekScheduledTask();  
  
    if (scheduledTask == null) {  
  
        return SCHEDULE_PURGE_INTERVAL;  
  
    }  
  
    return scheduledTask.delayNanos(currentTimeNanos);  
  
}
```

关于Netty的任务队列（包括普通任务、定时任务）相关的细节我们在接下来的小节会详细分析，这里就不过多展开了。

2.轮询过程中发现有任务加入，中断本次轮询。

```
for (;;) {  
  
    // 1.定时任务截止时间快到了，中断本次轮询  
  
    // 2.轮询过程中发现有任务加入，中断本次轮询  
  
    if (hasTasks() && wakenUp.compareAndSet(false, true)) {
```

```

    selector.selectNow();

    selectCnt = 1;

    break;

}

// 3.阻塞式select操作

// 4.解决JDK的NIO Bug

}

```

Netty为了保证任务队列里的任务能够及时执行，在进行阻塞select操作的时候会判断任务队列是否为空。如果不为空，就执行一次非阻塞select操作，跳出循环；否则，继续执行下面的操作。

3.阻塞式select操作。

```

for ( ; ; ) {

    // 1.定时任务截止时间快到了，中断本次轮询

    // 2.轮询过程中发现有任务加入，中断本次轮询

    // 3.阻塞式select操作

    int selectedKeys = selector.select(timeoutMillis);

    selectCnt++;

    if (selectedKeys != 0 || oldWakenUp || wakenUp.get() || hasTasks()
        ||
        hasScheduledTasks()) {

        break;

    }
}

```

```
// 4.解决JDK的NIO Bug  
}  
}
```

执行到这一步，说明Netty任务队列里的队列为空，并且所有定时任务的延迟时间还未到（大于0.5ms）。于是，进行一次阻塞select操作，截止到第一个定时任务的截止时间。

这里，我们可以问自己一个问题，如果第一个定时任务的延迟非常长，比如一小时，那么有没有可能线程一直阻塞在select操作？

答案是当然有可能，但是，只要在这段时间内有新任务加入，该阻塞就会被释放。我们接下来简单分析一下当有外部线程执行EventLoop的execute方法时会发生什么情况。

SingleThreadEventExecutor.java

```
@Override  
  
public void execute(Runnable task) {  
  
    ...  
  
    // inEventLoop 为 false, 所以调用这里的 wakeup 方法  
  
    wakeup(inEventLoop);  
  
    ...  
  
}
```

接下来进入wakeup的实现。

NioEventLoop.java

```
protected void wakeup(boolean inEventLoop) {  
  
    if (! inEventLoop && wakenUp.compareAndSet(false, true)) {
```

```
    selector.wakeup();

}

}
```

可以看到，在外部线程添加任务的时候，会调用wakeup方法来唤醒
selector.select(timeoutMillis)。

阻塞select操作结束之后，Netty又做了一系列状态判断来决定是否中断本次轮询，中断本次轮询的条件有如下几种，对应着if判断条件里面的一系列或条件。

- 检测到IO事件。
- 被用户主动唤醒。
- 任务队列里面有任务需要执行。
- 第一个定时任务即将要被执行。

4.解决JDK的NIO Bug。select操作的最后一步，就是解决JDK的NIO Bug。该Bug会导致Selector一直空轮询，最终导致CPU 100%，NIO Server不可用。严格意义上来说，Netty没有解决JDK的Bug，而是通过一种方式巧妙地避开了这个Bug，具体做法如下。

NioEventLoop.java

```
long currentTimeNanos = System.nanoTime();

for ( ; ; ) {

    // 1. 定时任务截止时间快到了，中断本次轮询

    long currentTimeNanos = System.nanoTime();

    // 2. 轮询过程中发现有任务加入，中断本次轮询

    // 3. 阻塞式select操作

    selector.select(timeoutMillis);

    // 4. 解决JDK的NIO Bug
```

```

long time = System.nanoTime();

if (time - TimeUnit.MILLISECONDS.toNanos(timeoutMillis) >=
currentTimeNanos) {

    selectCnt = 1;

} else if (SELECTOR_AUTO_REBUILD_THRESHOLD > 0 &&
selectCnt >= SELECTOR_AUTO_REBUILD_THRESHOLD) {

    rebuildSelector();

    selector = this.selector;

    selector.selectNow();

    selectCnt = 1;

    break;

}

currentTimeNanos = time;

...

}

```

Netty会在每次进行selector.select(timeoutMillis)之前记录下开始时间currentTimeNanos，在阻塞select之后记录下结束时间，判断select操作是否至少持续了timeoutMillis秒（这里将time-TimeUnit.MILLISECONDS.toNanos(timeoutMillis)>=currentTimeNanos改成time-currentTimeNanos >=TimeUnit.MILLISECONDS.toNanos(timeoutMillis)或许更好理解一些），如果持续的时间大于等于timeoutMillis，说明这是一次有效的轮询，重置selectCnt标志，否则，表明该阻塞方法并没有阻塞这么长时间，可能触发（说可能是因为还可能是外部线程执行任务中断了本次select操作）了JDK的空轮询Bug，当空轮询的次数超过一定阈值（默认是512）的时候，就开始重建Selector。

空轮询阈值的设置代码如下。

NioEventLoop.java

```
int selectorAutoRebuildThreshold =
SystemPropertyUtil.getInt("io.netty.

selectorAutoRebuildThreshold", 512);

if (selectorAutoRebuildThreshold < MIN_PREMATURE_SELECTOR RETURNS)  {

    selectorAutoRebuildThreshold = 0;

}

SELECTOR_AUTO_REBUILD_THRESHOLD = selectorAutoRebuildThreshold;
```

由此可以看到， 默认SELECTOR_AUTO_REBUILD_THRESHOLD为512。

下面我们简单描述一下Netty通过rebuildSelector来绕过空轮询Bug的过程。

rebuildSelector操作其实很简单：创建一个新的Selector， 将之前注册到老的Selector上的Channel重新转移到新的Selector上。我们抽取完主要代码之后的骨架如下。

NioEventLoop.java

```
public void rebuildSelector()  {

    final Selector oldSelector = selector;

    final Selector newSelector;

    newSelector = openSelector();

    int nChannels = 0;

    try  {

        for (； )  {

            for (SelectionKey key: oldSelector.keys())  {

                Object a = key.attachment();
```

```
// 1.拿到有效的key

if (! key.isValid() || key.channel().keyFor(newSelector) != null) {

continue;

}

int interestOps = key.interestOps();

// 2.取消该key在旧的Selector上的事件注册

key.cancel();

// 3.将该key对应的cancel注册到新的Selector上

SelectionKey newKey = key.channel().register(newSelector,
interestOps, a);

// 4.重新绑定Channel和新的key

if (a instanceof AbstractNioChannel) {

((AbstractNioChannel) a).selectionKey = newKey;

}

nChannels++;

}

break;

}

}

} catch (ConcurrentModificationException e) {

continue;

}

selector = newSelector;
```

```
oldSelector.close();

    }
```

首先，通过openSelector()方法创建一个新的Selector，然后执行一个无限for循环，只要执行过程中出现过一次并发修改SelectionKeys异常，就重新开始转移，直到转移完成。

具体的转移步骤如下。

- 1.拿到有效的key。
- 2.取消该key在旧的Selector上的事件注册。
- 3.将该key对应的Channel注册到新的Selector上。
- 4.重新绑定Channel和新的key。

转移完成之后，就可以将原有的Selector废弃，后面所有的轮询都在新的Selector上进行。

最后，我们总结Reactor线程select操作做的事情：不断地轮询是否有IO事件发生，并且在轮询过程中不断检查是否有任务需要执行，保证Netty任务队列中的任务能够及时执行，轮询过程使用一个计数器避开了JDK的空轮询Bug，整个过程还是比较清晰的。

22.3.3 处理产生IO事件的Channel

我们已经了解到Netty Reactor线程执行总体框架的第一步是轮询出注册在Selector上的IO事件，那么接下来就要处理这些IO事件（process selected keys）。下面我们一起探讨Netty处理IO事件的细节。

我们进入Reactor线程的run方法，找到处理IO事件的代码，如下所示。

NioEventLoop.java

```
processSelectedKeys();

private void processSelectedKeys() {

    if (selectedKeys != null) {
```

```
processSelectedKeysOptimized(selectedKeys.flip());  
}  
else {  
    processSelectedKeysPlain(selector.selectedKeys());  
}  
}  
}
```

我们发现处理IO事件，Netty有两种选择，从名字上看，一种是处理优化过的SelectedKeys，一种是正常处理。

我们把对优化过的SelectedKeys的处理稍微展开一下，看看Netty是如何优化的。我们查看SelectedKeys被引用过的地方，有如下代码。

NioEventLoop.java

```
private SelectedSelectionKeySet selectedKeys;  
  
private Selector NioEventLoop.openSelector() {  
    //...  
  
    final SelectedSelectionKeySet selectedKeySet = new  
    SelectedSelectionKeySet();  
  
    // selectorImplClass -> sun.nio.ch.SelectorImpl  
  
    Field selectedKeysField =  
    selectorImplClass.getDeclaredField("selectedKeys");  
  
    Field publicSelectedKeysField =  
    selectorImplClass.getDeclaredField("publicSelectedKeys");  
  
    selectedKeysField.setAccessible(true);  
  
    publicSelectedKeysField.setAccessible(true);  
  
    selectedKeysField.set(selector, selectedKeySet);
```

```
publicSelectedKeysField.set(selector, selectedKeySet);

//...

selectedKeys = selectedKeySet;

}
```

首先，SelectedKeys是一个SelectedSelectionKeySet类对象，在NioEventLoop的openSelector方法中创建，之后通过反射将SelectedKeys与sun.nio.ch.SelectorImpl中的两个成员变量绑定。

在sun.nio.ch.SelectorImpl中，我们可以看到，其实这两个成员变量是两个HashSet。

SelectorImpl.java

```
// Public views of the key sets

private Set<SelectionKey> publicKeys;

private Set<SelectionKey> publicSelectedKeys;

protected SelectorImpl(SelectorProvider sp)  {

    super(sp);

    keys = new HashSet<SelectionKey>();

    selectedKeys = new HashSet<SelectionKey>();

    if (Util.atBugLevel("1.4"))  {

        publicKeys = keys;

        publicSelectedKeys = selectedKeys;

    } else {

        publicKeys = Collections.unmodifiableSet(keys);

    }

}
```

```
publicSelectedKeys = Util.ungrowableSet(selectedKeys);

    }

}
```

Selector在调用select()族方法的时候，如果有IO事件发生，就会往里面的两个成员变量中塞相应的SelectionKey，即相当于往HashSet中添加元素，既然Netty通过反射将JDK中的两个成员变量替换掉，那我们就应该意识到，是不是SelectedSelectionKeySet在add方法中做了某些优化呢？

带着这个疑问，我们进入SelectedSelectionKeySet类探个究竟。

SelectedSelectionKeySet.java

```
private SelectionKey [] keysA;

private int keysASize;

private SelectionKey [] keysB;

private int keysBSize;

private boolean isA = true;

@Override

public boolean add(SelectionKey o) {

    if (o == null) {

        return false;

    }

    if (isA) {

        int size = keysASize;

        keysA [size ++] = o;

        keysASize = size;

    }

}
```

```

    if (size == keysA.length) {

        doubleCapacityA();

    }

} else {

    int size = keysBSize;

    keysB [size ++] = o;

    keysBSize = size;

    if (size == keysB.length) {

        doubleCapacityB();

    }

}

return true;

}

```

这个类继承了AbstractSet，说明该类可以当作一个set来用，但是在底层使用两个数组来交替使用。在add方法中，判断当前使用哪个数组，找到对应的数组，然后经历下面3个步骤：（1）将SelectionKey塞到该数组的尾部；（2）更新该数组的逻辑长度+1；（3）如果该数组的逻辑长度等于数组的物理长度，就将该数组扩容。

可以看到，待程序运行一段时间后，等数组的长度足够长，每次在轮询到NIO事件的时候，Netty只需要 $O(1)$ 的时间复杂度就能将SelectionKey塞到set中去，而JDK底层使用的HashSet put的时间复杂度最少是 $O(1)$ ，最差是 $O(n)$ ，使用数组替换掉HashSet还有一个好处是遍历的时候非常高效。

这里关于为何使用两个数组循环交替，其实笔者也很费解。查找所有使用SelectedSelection-KeySet的地方，笔者觉得使用一个数组就能够达到优化目的，并且不用每次都判断使用哪个数组，所以对于该问题，笔者提了一个Issue给Netty官方，目前在4.1.9.Final版本中，Netty已经在SelectedSelectionKeySet.java底层使用一个数组了。

关于Netty对SelectionKeySet的优化，我们暂时就分析这么多。下面我们继续分析Netty对IO事件的处理，转到processSelectedKeysOptimized。

NioEventLoop.java

```
private void processSelectedKeysOptimized(SelectionKey []  
selectedKeys)  {  
  
    for (int i = 0; ; i++)  {  
  
        // 1.取出 IO 事件及对应的 Channel  
  
        final SelectionKey k = selectedKeys [i] ;  
  
        if (k == null)  {  
  
            break;  
  
        }  
  
        selectedKeys [i]  = null;  
  
        final Object a = k.attachment();  
  
        // 2.处理该Channel  
  
        if (a instanceof AbstractNioChannel)  {  
  
            processSelectedKey(k, (AbstractNioChannel) a);  
  
        } else {  
  
            NioTask<SelectableChannel> task = (NioTask<SelectableChannel>) a;  
  
            processSelectedKey(k, task);  
  
        }  
  
        // 3.判断是否该再进行一次轮询  
  
        if (needsToSelectAgain)  {
```

```
for ( ; ; ) {  
  
    i++;  
  
    if (selectedKeys [i] == null) {  
  
        break;  
  
    }  
  
    selectedKeys [i] = null;  
  
}  
  
selectAgain();  
  
selectedKeys = this.selectedKeys.flip();  
  
i = -1;  
  
}  
  
}  
  
}
```

我们可以将该过程分为以下3个步骤。

1.取出IO事件及对应的Channel。这里其实也能体会到优化过的
SelectedSelectionKeySet的好处，遍历时遍历的是数组，相对JDK原生的HashSet，效
率有所提高。

拿到当前的SelectionKey之后，将selectedKeys[i]置为null。这里简单解释一下这么做
的理由：想象一下这种场景，假设一个NioEventLoop平均每次轮询出 N 个IO事件，高
峰期轮询出 $3N$ 个事件，那么SelectedKeys的物理长度要大于等于 $3N$ ，如果每次处理
这些key，不置selectedKeys[i]为空，那么高峰期一过，这些保存在数组尾部的
selectedKeys[i]对应的SelectionKey将一直无法被回收，SelectionKey对应的对象可能
不大，但是要知道，它可是有attachment的。这里的attachment具体是什么下面会讲
到，但是有一点我们必须清楚，attachment可能很大，这样一来，这些对象就一直存
活，造成JVM无法回收，内存泄漏就发生了。

2. 处理该Channel。拿到对应的attachment之后，Netty做了如下判断。

NioEventLoop.java

```
if (a instanceof AbstractNioChannel) {  
  
    processSelectedKey(k, (AbstractNioChannel) a);  
  
}
```

源码读到这，我们需要思考为什么会有这么一条判断，为什么说attachment可能会是AbstractNioChannel对象？这个问题其实在上一章中已经有了答案，我们再来分析一下。

我们的思路应该是找到底层Selector，然后在Selector调用register方法的时候，看一下注册到Selector上的对象是什么，我们在AbstractNioChannel中搜索到如下方法。

AbstractNioChannel.java

```
protected void doRegister() throws Exception {  
  
    // ...  
  
    selectionKey = javaChannel().register(eventLoop().selector, 0,  
    this);  
  
    // ...  
  
}
```

javaChannel方法返回Netty类AbstractChannel对应的JDK底层Channel对象。

AbstractNioChannel.java

```
protected SelectableChannel javaChannel() {  
  
    return ch;  
  
}
```

我们查看到SelectableChannel方法，结合Netty的doRegister()方法，不难推论出，Netty的轮询注册机制其实是将AbstractNioChannel内部的JDK类SelectableChannel对象注册到JDK类Selector对象上，并且将AbstractNioChannel作为SelectableChannel对象的一个attachment附属上，这样在JDK轮询出某条SelectableChannel有IO事件发生时，就可以直接取出AbstractNioChannel进行后续操作。

在Netty的Channel中，有两大类型的Channel，一个是NioServerSocketChannel，由boss NioEventLoopGroup负责处理；一个是NioSocketChannel，由worker NioEventLoop负责处理，所以：

- (1) 对于boss NioEventLoop来说，轮询到的是连接事件，后续通过NioServerSocketChannel的Pipeline将连接交给一个worker NioEventLoop处理；
- (2) 对于worker NioEventLoop来说，轮询到的是读写事件，后续通过NioSocketChannel的Pipeline将读取到的数据传递给每个ChannelHandler来处理。

这两部分逻辑体现在如下方法中。

NioEventLoop.java

```
private void processSelectedKey(SelectionKey k, AbstractNioChannel ch)
{
    int readyOps = k.readyOps();

    if (((readyOps & (SelectionKey.OP_READ | SelectionKey.OP_ACCEPT)) != 0 || readyOps
        == 0) {
        unsafe.read();
    }
}
```

我们分析新连接接入的时候，这里的代码就是入口。

processSelectedKeysOptimized方法处理attachment的时候，还有一个else分支，我们也来分析一下。

NioEventLoop.java

```
NioTask<SelectableChannel> task = (NioTask<SelectableChannel>) a;  
  
processSelectedKey(k, task);
```

说明注册到Selector上的attachment还有另外一种类型，就是NioTask， NioTask主要用于当一个SelectableChannel注册到Selector的时候，执行一些任务。

NioTask的定义如下。

```
public interface NioTask<C extends SelectableChannel> {  
  
    void channelReady(C ch, SelectionKey key) throws Exception;  
  
    void channelUnregistered(C ch, Throwable cause) throws Exception;  
  
}
```

由于NioTask在Netty内部没有使用的地方，这里不过多展开介绍。

3.判断是否该再进行一次轮询。

NioEventLoop.java

```
if (needsToSelectAgain) {  
  
    for ( ; ; ) {  
  
        i++;  
  
        if (selectedKeys [i] == null) {  
  
            break;  
  
        }  
  
        selectedKeys [i] = null;  
  
    }  
  
    selectAgain();
```

```
selectedKeys = this.selectedKeys.flip();

i = -1;

}
```

我们回忆一下Netty Reactor线程的前两个步骤，分别是抓取IO事件及处理IO事件。每次在抓取到IO事件之后，都会将needsToSelectAgain重置为false，那么什么时候needsToSelectAgain会重新被设置成true呢？

我们使用IDE来帮助查看needsToSelectAgain被使用的地方，在NioEventLoop类中，只有下面一处将needsToSelectAgain设置为true。

NioEventLoop.java

```
void cancel(SelectionKey key) {

key.cancel();

cancelledKeys++;

if (cancelledKeys >= CLEANUP_INTERVAL) {

cancelledKeys = 0;

needsToSelectAgain = true;

}

}

}
```

继续查看cancel方法被调用的地方。

AbstractChannel.java

```
@Override

protected void doDeregister() throws Exception {

eventLoop().cancel(selectionKey());

}

}
```

不难看出，Channel从Selector上移除的时候，调用cancel方法将key取消，并且在被取消的key到达CLEANUP_INTERVAL的时候，设置needsToSelectAgain为true，CLEANUP_INTERVAL默认值为256。

NioEventLoop.java

```
private static final int CLEANUP_INTERVAL = 256;
```

也就是说，对于每个NioEventLoop而言，每隔256个Channel从Selector上移除的时候，就标记needsToSelectAgain为true，我们还是跳回到上面这段代码。

NioEventLoop.java

```
if (needsToSelectAgain) {  
  
    for ( ; ; ) {  
  
        i++;  
  
        if (selectedKeys [i] == null) {  
  
            break;  
  
        }  
  
        selectedKeys [i] = null;  
  
    }  
  
    selectAgain();  
  
    selectedKeys = this.selectedKeys.flip();  
  
    i = -1;  
  
}
```

每满256次，就会进入if代码块。首先，将SelectedKeys的内部数组全部清空，方便JVM垃圾回收，然后调用selectAgain重新填装SelectionKeys数组。

NioEventLoop.java

```
private void selectAgain() {  
    needsToSelectAgain = false;  
    selector.selectNow();  
}
```

Netty这么做的目的应该是每隔256次连接断开，重新清理一下SelectionKeys，这相当于用批量删除替代了Selector原HashSet数据结构的删除，从SelectedSelectionKeySet的remove方法和contains方法的实现就可以看出来。

SelectedSelectionKeySet.java

```
@Override  
  
public boolean remove(Object o) {  
    return false;  
}  
  
@Override  
  
public boolean contains(Object o) {  
    return false;  
}
```

Selector原始的HashSet在put操作和remove操作的时候，最坏情况下的时间复杂度都为 $O(n)$ ，而Netty使用数组替换底层的数据结构之后，两者的时间复杂度都降到 $O(1)$ 。

最后，我们对处理IO事件部分的内容总结一下：Netty Reactor线程第二步做的事情为处理IO事件。

1. Netty使用数组替换JDK原生的HashSet来提升处理IO事件的效率。
2. 每个SelectionKey上都绑定了Netty类AbstractChannel对象作为attachment，在处理每个SelectionKey的时候，都可以找到AbstractChannel，然后通过Pipeline将处理串行

到ChannelHandler，回调到用户方法。

22.3.4 添加任务

前面我们已经知道，每一次事件轮询，首先都会检测出IO事件，如果有IO事件，那么就去处理。IO事件主要包含新连接接入事件和连接的数据读写事件，这两步处理完之后，还有最后一步：处理任务队列，即runAllTasks()。

这一部分，我们先分析在用户代码里添加任务的逻辑，在第22.3.5节再分析任务的执行。我们取3种典型的Task使用场景来分析。

1. 用户自定义普通任务。

```
ctx.channel().eventLoop().execute(new Runnable() {
    @Override
    public void run() {
        //...
    }
});
```

我们跟进execute方法。

SingleThreadEventExecutor.java

```
@Override
public void execute(Runnable task) {
    boolean inEventLoop = inEventLoop();
    if (inEventLoop) {
        addTask(task);
    } else {
```

```
    addTask(task);

    }

}

}
```

我们看到，不管是外部线程还是Reactor线程，execute方法都会调用addTask方法。

SingleThreadEventExecutor.java

```
protected void addTask(Runnable task) {

    // ...

    if (! offerTask(task)) {

        reject(task);

    }

}

}
```

然后调用offerTask方法，如果失败，那就调用reject方法，通过默认的RejectedExecutionHandler直接抛出异常。

SingleThreadEventExecutor.java

```
final boolean offerTask(Runnable task) {

    // ...

    return taskQueue.offer(task);

}

}
```

我们跟到offerTask方法，这个Task就落地了，Netty内部使用一个taskQueue将Task保存起来。我们在第22.1.4节已经分析过这个taskQueue，它其实是一个MPSC Queue，每一个NioEventLoop都与它一一对应。

Netty使用MPSC Queue，方便将外部线程的异步任务聚集，在Reactor线程内部用单线程来批量执行以提升性能。我们可以借鉴Netty的任务执行模式来处理类似多线程

数据聚合，定时上报应用。

2.非当前Reactor线程调用Channel的各类方法。下面这个例子，在代码里经常会出
现。

```
// 当前线程为业务线程  
  
channel.write(...)
```

这个场景是：在业务线程里，根据用户的标识，找到对应的Channel，然后调用
Channel的write类方法向该用户推送消息。

关于channel.write方法的调用，后面会详细剖析。这里我们只需要知道，最终write方
法调用以下方法。

AbstractChannelHandlerContext.java

```
private void write(Object msg, boolean flush, ChannelPromise promise)  
{  
  
    // ...  
  
    EventExecutor executor = next.executor();  
  
    if (executor.inEventLoop()) {  
  
        if (flush) {  
  
            next.invokeWriteAndFlush(m, promise);  
  
        } else {  
  
            next.invokeWrite(m, promise);  
  
        }  
    } else {  
  
        AbstractWriteTask task;  
  
        if (flush) {  
            task = new WriteAndFlushTask(next, m, promise);  
        } else {  
            task = new WriteTask(next, m, promise);  
        }  
        executor.execute(task);  
    }  
}  
  
public void write(Object msg, ChannelPromise promise)  
{  
    write(msg, false, promise);  
}  
  
public void write(Object msg)  
{  
    write(msg, false);  
}  
  
public void write()  
{  
    write(null);  
}
```

```

task = WriteAndFlushTask.newInstance(next, m, promise);

} else {

task = WriteTask.newInstance(next, m, promise);

}

safeExecute(executor, task, promise, m);

}

}

```

外部线程在调用write的时候，executor.inEventLoop()会返回false，接下来进入else分支，将write操作封装成一个WriteTask（这里仅仅是write而没有flush，因此flush参数为false），然后调用safeExecute方法来执行。

AbstractEventExecutor.java

```

private static void safeExecute(EventExecutor executor, Runnable
runnable,

ChannelPromise promise, Object msg) {

// ...

executor.execute(runnable);

// ...

}

```

接下来的调用链就进入第一种场景了，但是和第一种场景有个明显的区别就是，第一种场景的调用链的发起线程是Reactor线程，第二种场景的调用链的发起线程是用户线程，用户线程可能会有很多个，在这种场景下，Netty的MPSC Queue就有了用武之地。

3. 用户自定义定时任务。第三种场景就是定时任务逻辑了，用得最多的便是如下方法。

AbstractScheduledEventExecutor.java

```
ctx.channel().eventLoop().schedule(new Runnable() {  
  
    @Override  
  
    public void run() {  
  
    }  
  
}, 60, TimeUnit.SECONDS);
```

这段代码的作用是，在一定时间之后执行某个任务。

我们跟进schedule方法。

AbstractScheduledEventExecutor.java

```
public ScheduledFuture<?> schedule(Runnable command, long delay,  
TimeUnit unit) {  
  
//...  
  
return schedule(new ScheduledFutureTask<Void>(  
  
this, command, null,  
ScheduledFutureTask.deadlineNanos(unit.toNanos(delay))));  
  
}
```

通过ScheduledFutureTask，将用户自定义任务再次包装成一个Netty内部的任务。

AbstractScheduledEventExecutor.java

```
<V> ScheduledFuture<V> schedule(final ScheduledFutureTask<V> task) {  
  
// ...  
  
scheduledTaskQueue().add(task);  
  
// ...
```

```
    return task;

}
```

到了这里，对于scheduledTaskQueue，我们觉得有点似曾相识，在非定时任务的处理中，Netty通过一个MPSC队列将任务落地。这里，是否也有一个类似的队列来承载这类定时任务呢？带着这个疑问，我们继续向前探索。

AbstractScheduledEventExecutor.java

```
Queue<ScheduledFutureTask<?>> scheduledTaskQueue() {

    if (scheduledTaskQueue == null) {

        scheduledTaskQueue = new PriorityQueue<ScheduledFutureTask<?>>();

    }

    return scheduledTaskQueue;

}
```

果不其然，scheduledTaskQueue()方法会返回一个优先级队列，然后调用add方法将定时任务对象ScheduledFutureTask加入队列，但是，为什么可以直接使用优先级队列，而不需要考虑多线程的并发？

如果是在外部线程调用schedule方法，Netty会将添加定时任务这个逻辑封装成一个普通的task，这个task的任务是一个添加“添加定时任务”的任务，而不是添加定时任务，所以其实就退回到第二种场景。这样，对PriorityQueue的访问就变成单线程，即只有Reactor线程会访问，因此，不存在多线程并发问题。

以下就是如何保证定时任务并发完整的逻辑。

AbstractScheduledEventExecutor.java

```
<V> ScheduledFuture<V> schedule(final ScheduledFutureTask<V> task) {

    // 如果是Reactor线程，则直接往优先级队列中添加任务

    if (inEventLoop()) {


```

```
scheduledTaskQueue().add(task);

} else {

// 如果是外部线程，则将添加定时任务这个逻辑进一步封装，退回到场景二

execute(new Runnable() {

@Override

public void run() {

scheduledTaskQueue().add(task);

}

}

);

}

return task;

}
```

在阅读源码的过程中，我们应该多问几个为什么。比如这里，为什么定时任务要保存在优先级队列中，我们可以先不看源码，来思考一下优先级队列的特性：优先级队列按照一定顺序来排列内部元素，内部元素必须是可以比较的，联系到这里的每个元素都是定时任务，那就说明定时任务是可以比较的，那么到底有哪些地方可以比较呢？

每个任务都有一个下一次执行的截止时间，截止时间是可以比较的。在截止时间相同的情况下，任务添加的顺序也是可以比较的。就像这样，在阅读源码的过程中，一定要多和自己对话，多问几个为什么。

带着猜想，我们研究一下ScheduledFutureTask，抽取出关键代码部分。

ScheduledFutureTask.java

```
final class ScheduledFutureTask<V> extends PromiseTask<V> implements
ScheduledFutureTask<V>

{
```

```
// 每个任务都有一个唯一的 ID

private static final AtomicLong nextTaskId = new AtomicLong();

private static final long START_TIME = System.nanoTime();

static long nanoTime() {

    return System.nanoTime() - START_TIME;

}

private final long id = nextTaskId.getAndIncrement();

// 标识一个任务是否重复执行及以何种方式来定期执行

private final long periodNanos;

@Override

public int compareTo(Delayed o) {

//...

}

// 精简过的代码

@Override

public void run() {

}
```

这里，我们一眼就找到了compareTo方法，这个方法实现自Comparable接口。接下来，我们分析一下这个方法的实现。

```
public int compareTo(Delayed o) {

if (this == o) {
```

```
    return 0;

}

ScheduledFutureTask<?> that = (ScheduledFutureTask<?>) o;

long d = deadlineNanos() - that.deadlineNanos();

if (d < 0) {

    return -1;

} else if (d > 0) {

    return 1;

} else if (id < that.id) {

    return -1;

} else if (id == that.id) {

    throw new Error();

} else {

    return 1;

}

}
```

进入方法体内部，我们发现，两个定时任务的比较，确实是先比较任务的截止时间；在截止时间相同的情况下，再比较ID，即任务添加的顺序；如果ID再相同，就抛出Error。

这样，在执行定时任务的时候，就能保证截止时间最近的任务先执行。

Netty里的定时任务机制，除了我们前面提到的schedule方法，还有以下两种。

1.scheduleAtFixedRate：每隔一段时间执行一次。

2.scheduleWithFixedDelay：隔相同时间再执行一次。

对于这3种方法，调用的时候逻辑非常类似，唯一的区别就是，在构造ScheduledFutureTask的时候，某个参数不一样，这个参数就是periodNanos。

ScheduledFutureTask.java

```
/* 0 - no repeat, >0 - repeat at fixed rate, <0 - repeat with fixed
delay */
```

```
private final long periodNanos;
```

1.schedule方法：传递的periodNanos为0，表示这个任务不会被重复执行。

2.scheduleAtFixedRate方法：传递的periodNanos为正数，表示以固定速度来执行任务，与任务执行的耗时无关。

3.scheduleWithFixedDelay：传递的periodNanos为负数，表示以固定的延时来执行任务，即每次任务执行完毕之后，隔相同的时间再次执行。

下面一段代码，就是Netty处理这3种定时任务的逻辑。

ScheduledFutureTask.java

```
public void run() {
    // 1. 对应 schedule 方法，表示一次性任务
    if (periodNanos == 0) {
        V result = task.call();
        setSuccessInternal(result);
    } else {
        task.call();
        long p = periodNanos;
        // 2. 对应 scheduleAtFixedRate 方法，表示以固定速度执行任务
    }
}
```

```
if (p > 0) {  
  
    deadlineNanos += p;  
  
} else {  
  
    // 3. 对应 scheduleWithFixedDelay 方法, 表示以固定的延时执行任务, 这里是 p  
    // 为负数,  
}
```

相当于加上一个正数

```
deadlineNanos = nanoTime() - p;
```

```
}
```

```
scheduledTaskQueue.add(this);
```

```
}
```

```
}
```

```
}
```

if(periodNanos==0)对应若干时间后执行一次的定时任务类型，执行完了该任务就结束了。

否则，进入else代码块，先执行任务，再区分是哪种类型的任务。

1.如果periodNanos大于0，表示以固定速度执行某个任务，和任务的持续时间无关，所以该任务下一次执行的截止时间为本次截止时间加上间隔时间-p。

2.如果periodNanos小于0，表示每次任务执行完毕之后，间隔多长时间之后再次执行，所以该任务下一次执行的截止时间为当前时间加上间隔时间，-p就表示加上一个正的间隔时间。

其实这里可以看出，Netty的3种定时任务逻辑是通过调整下一次任务的截止时间来运行的，修改完下一次执行的截止时间，把当前任务再次加入队列，就能够确保任务的适时执行。

Netty内部的任务添加机制了解得差不多之后，我们就可以分析Netty Reactor线程执行总体框架的第三个步骤，即runAllTasks了。

22.3.5 执行任务

首先，我们将目光转向最外层的外观代码。

SingleThreadEventExecutor.java

```
runAllTasks(long timeoutNanos);
```

顾名思义，这行代码表示尽量在一定时间内将所有的任务都取出来执行一遍。
timeoutNanos表示该方法最多执行多长时间。

Netty为什么要这么做？我们可以想一想，Reactor线程如果在此停留的时间过长，那么将积攒许多的IO事件无法处理（见Reactor线程的前两个步骤），最终导致大量客户端请求阻塞。因此，在默认情况下，Netty会精细控制内部任务队列的执行时间。

接下来详细分析任务执行逻辑，我们依然会抽取出关键代码。

SingleThreadEventExecutor.java

```
protected boolean runAllTasks(long timeoutNanos) {  
  
    // 1. 转移定时任务至MPSC Queue  
  
    fetchFromScheduledTaskQueue();  
  
    Runnable task = pollTask();  
  
    // 2. 计算本轮任务执行的截止时间  
  
    final long deadline = ScheduledFutureTask.nanoTime() + timeoutNanos;  
  
    long runTasks = 0;  
  
    long lastExecutionTime;  
  
    // 3. 执行任务  
  
    for (; ; ) {  
  
        safeExecute(task);  
  
        runTasks++;
```

```
if ((runTasks & 0x3F) == 0) {  
  
    lastExecutionTime = ScheduledFutureTask.nanoTime();  
  
    if (lastExecutionTime >= deadline) {  
  
        break;  
  
    }  
  
}  
  
task = pollTask();  
  
if (task == null) {  
  
    lastExecutionTime = ScheduledFutureTask.nanoTime();  
  
    break;  
  
}  
  
}  
  
}  
  
this.lastExecutionTime = lastExecutionTime;  
  
return true;  
  
}
```

以上这段代码便是Reactor执行任务的逻辑，可以拆解成下面3个步骤。

- 1.从定时任务队列scheduledTaskQueue转移定时任务到普通任务队列taskQueue。
- 2.计算本轮任务执行的截止时间。
- 3.执行任务。

按照这3个步骤，我们一步步来分析下。

- 1.转移定时任务至MPSC Queue。首先调用fetchFromScheduledTaskQueue()方法，将快到期的定时任务转移到MPSC Queue里面。

SingleThreadEventExecutor.java

```
protected boolean runAllTasks(long timeoutNanos)  {

    // 1. 转移定时任务至MPSC Queue

    fetchFromScheduledTaskQueue();

    // 2. 计算本轮任务执行的截止时间

    ...

    // 3. 执行任务

    ...

}

private boolean fetchFromScheduledTaskQueue()  {

    long nanoTime = AbstractScheduledEventExecutor.nanoTime();

    Runnable scheduledTask = pollScheduledTask(nanoTime);

    while (scheduledTask != null)  {

        if (! taskQueue.offer(scheduledTask))  {

            scheduledTaskQueue().add((ScheduledFutureTask<?>) scheduledTask);

        }

        return false;

    }

    scheduledTask = pollScheduledTask(nanoTime);

}

return true;

}
```

可以看到，Netty在把任务从scheduledTaskQueue转移到taskQueue的时候还是非常小心的。当taskQueue offer失败的时候，需要把从scheduledTaskQueue里取出来的任务重新添加回去。

从scheduledTaskQueue中拉取一个定时任务的逻辑如下，传入的参数nanoTime为当前纳秒减去ScheduledFutureTask类被加载时的纳秒。

AbstractScheduledEventExecutor

```
protected static long nanoTime() {  
  
    return ScheduledFutureTask.nanoTime();  
  
}
```

nanoTime()的计算方法如下。

ScheduledFutureTask.java

```
private static final long START_TIME = System.nanoTime();  
  
static long nanoTime() {  
  
    return System.nanoTime() - START_TIME;  
  
}
```

这个时间就表示当前时间相对ScheduledFutureTask类加载的时间。前面关于ScheduledFutureTask，其实我们没有展开介绍，在创建ScheduledFutureTask的时候，deadlineNanos也被设置为相对ScheduledFutureTask类加载的时间，感兴趣的读者可以详细分析一下第22.3.4节的3类schedule方法中ScheduledFutureTask的构造逻辑。

Netty使用当前的相对时间与任务的相对截止时间进行比较。

SingleThreadEventExecutor.java

```
protected final Runnable pollScheduledTask(long nanoTime) {  
  
    assert inEventLoop();
```

```

    Queue<ScheduledFutureTask<?>> scheduledTaskQueue =
this.scheduledTaskQueue;

    ScheduledFutureTask<?> scheduledTask = scheduledTaskQueue == null ?
null :

scheduledTaskQueue.peek();

    if (scheduledTask == null) {

return null;

    }

// 任务的相对截止时间与当前的相对时间比较

    if (scheduledTask.deadlineNanos() <= nanoTime) {

scheduledTaskQueue.remove();

return scheduledTask;

    }

return null;

}

```

可以看到，只有在当前任务的截止时间已经到了时，该任务才会被移出队列。

2.计算本轮任务执行的截止时间。到了这一步，所有截止时间已经到达的定时任务均被填充到普通任务队列。接下来，Netty会计算一下本轮任务最多可以执行到什么时候。

SingleThreadEventExecutor.java

```

protected boolean runAllTasks(long timeoutNanos) {

// 1. 转移定时任务至MPSC Queue

// 2. 计算本轮任务执行的截止时间

```

```
final long deadline = ScheduledFutureTask.nanoTime() + timeoutNanos;

long runTasks = 0;

long lastExecutionTime;

// 3. 执行任务

...

}
```

Netty使用Reactor线程传入的超时时间timeoutNanos来计算当前任务循环的截止时间，并且使用runTasks、lastExecutionTime来时刻记录任务的状态。

3.执行任务。

```
protected boolean runAllTasks(long timeoutNanos) {

    // 1. 转移定时任务至MPSC Queue

    // 2. 计算本轮任务执行的截止时间

    // 3. 执行任务

    for (; ; ) {

        // 3.1 不抛异常则执行任务

        safeExecute(task);

        // 3.2 累计当前已执行任务

        runTasks++;

        // 3.3 每隔 64 次计算当前时间是否已过截止时间

        if ((runTasks & 0x3F) == 0) {

            lastExecutionTime = ScheduledFutureTask.nanoTime();

            if (lastExecutionTime >= deadline) {


```

这一步便是Netty里面执行所有任务的核心代码了。首先调用safeExecute来确保任务安全执行，忽略任何异常。

AbstractEventExecutor.java

```
protected static void safeExecute(Runnable task)  {  
  
    try  {  
  
        task.run();  
  
    } catch (Throwable t)  {  
  
        logger.warn("A task raised an exception. Task:  {}", task, t);  
  
    }  
  
}  
}
```

然后，将已运行任务runTasks加一。接着，每隔0x3F（64）个任务，判断当前时间是否超过本次Reactor任务循环的截止时间。如果超过，那就停止本轮任务的执行；如果没有超过，那就继续执行。最后，如果任务全部执行完毕，则记录下最后一次任务执行时间。

我们可以看到，Netty对性能的优化考虑得相当周到：假设任务队列里有海量小任务，如果每次执行完任务都要判断是否到截止时间，那么效率是比较低的，而通过批量的方式，效率要高很多。

Netty很多性能优化用的都是批量策略，我们在后面的章节中还会遇到。

22.3.6 NioEventLoop 的执行流程小结

- 1.NioEventLoop在执行过程中不断检测是否有事件发生，如果有事件发生就处理，处理完事件之后再处理外部线程提交过来的异步任务。
- 2.在检测是否有事件发生的时候，为了保证异步任务的及时处理，只要有任务要处理，就立即停止事件检测，随即处理任务。
- 3.外部线程异步执行的任务分为两种：定时任务和普通任务，分别落地到MpscQueue和PriorityQueue，而PriorityQueue中的任务最终都会填充到MpscQueue中处理。
- 4.Netty每隔64个任务检查一次是否该退出任务循环。

22.4 总结

最后，我们对本章内容进行总结。

- 1.NioEventLoopGroup在用户代码中被创建，默认情况下会创建两倍CPU核数个NioEventLoop。
- 2.NioEventLoop是懒启动的，boss NioEventLoop在服务端启动的时候启动，worker NioEventLoop在新连接接入的时候启动。
- 3.当CPU核数为2的幂时，为每一个新连接绑定NioEventLoop之后都会做一个或转与的优化。
- 4.每个连接都对应一个Channel，每个Channel都绑定唯一一个NioEventLoop，每个NioEventLoop都对应一个FastThreadLocalThread线程实体和一个Selector。因此，单个连接的所有操作都在一个线程上执行，是线程安全的。

5. 每个NioEventLoop都对应一个Selector，这个Selector可以批量处理注册到它上面的Channel。

6. 每个NioEventLoop的执行过程都包括事件检测、处理，以及异步任务的执行。

7. 用户线程池在对Channel进行一些操作的时候，均为线程安全的，这是因为Netty会把外部线程的操作都封装成一个Task塞到这个Channel绑定的NioEventLoop中的MpscQueue，在该NioEventLoop事件循环的第三个过程中进行串行执行。

第23章

客户端连接接入流程解析

本章，我们来分析每个新连接在接入过程中，Netty底层的机制是如何实现的。

读本章之前，最好已经理解前面介绍的Reactor线程模型和Netty服务端的启动流程，下面我们简要回顾一下。

首先是Netty中的Reactor线程模型。

Netty中最核心的东西莫过于两种类型的Reactor线程。这两种类型的Reactor线程可以看作Netty中的两组发动机，驱动着Netty整个框架的运转。

一种类型是boss线程，专门用来接收新连接，然后封装成Channel对象传递给worker线程；还有一种类型是worker线程，专门用来处理连接上数据的读写。

不管是boss线程还是worker线程，所做的事情均分为以下3个步骤。

1. 轮询注册在Selector上的IO事件。

2. 处理IO事件。

3. 执行异步Task。

对于boss线程来说，第一步轮询出来的基本都是ACCEPT事件，表示有新的连接；而worker线程轮询出来的基本都是read或write事件，表示网络的读写事件。

其次是服务端启动流程。

服务端是在用户线程中开启的，通过bind方法，在第一次添加异步任务的时候启动boss线程。启动之后，当前服务器就可以开启监听。

23.1 新连接接入的总体流程

简单来说，新连接的接入流程可以分为3个过程。

1. 检测到有新连接。

2. 将新连接注册到worker线程。

3. 注册新连接的读事件。

接下来，我们详细分析这3个过程。

23.2 检测到有新连接

我们已经知道，当调用bind方法启动服务端之后，服务端的Channel，即NioServerSocketChannel，已经注册到boss Reactor线程，Reactor线程不断检测是否有新的事件，直到检测出有ACCEPT事件发生。

NioEventLoop.java

```
private void processSelectedKey(SelectionKey k, AbstractNioChannel ch)
{
    final AbstractNioChannel.NioUnsafe unsafe = ch.unsafe();

    int readyOps = k.readyOps();

    if ((readyOps & (SelectionKey.OP_READ | SelectionKey.OP_ACCEPT)) != 0 || readyOps
        == 0) {
        unsafe.read();
    }
}
```

上面这段代码是Reactor线程在第二个过程做的事情，表示boss Reactor线程已经轮询到SelectionKey.OP_ACCEPT事件，即表明有新连接进入，此时将调用Channel的Unsafe来进行实际的操作。

在服务端启动流程解析章节中，我们已经知道，服务端对应的Channel的Unsafe是NioMessageUnsafe，我们进入它的read方法，进入新连接处理的第二步。

23.3 注册Reactor线程

NioMessageUnsafe.java

```
private final List<Object> readBuf = new ArrayList<Object>();

public void read()  {

    assert eventLoop().inEventLoop();

    final ChannelPipeline pipeline = pipeline();

    final RecvByteBufAllocator.Handle allocHandle =
unsafe().recvBufAllocHandle();

    do  {

        // 1.创建 NioSocketChannel

        int localRead = doReadMessages(readBuf);

        if (localRead == 0)  {

            break;

        }

    } while (allocHandle.continueReading());

    // 2.设置并绑定 NioSocketChannel

    int size = readBuf.size();

    for (int i = 0;  i < size;  i ++)  {

        pipeline.fireChannelRead(readBuf.get(i));

    }

    readBuf.clear();

    pipeline.fireChannelReadComplete();

}

}
```

笔者省去了非关键部分的代码，可以看到，一上来，就用一条断言确定该read方法必须来自Reactor线程调用，然后获得Channel对应的Pipeline和RecvByteBufAllocator.Handle（这里先不展开介绍）。

接下来，调用doReadMessages方法不断地读取消息，用readBuf作为容器。其实读者可以猜到这里读取的是一个个连接，然后使用for循环调用pipeline.fireChannelRead()，将每个新连接都经过一层服务端Channel的Pipeline逻辑处理，之后清理容器，触发pipeline.fireChannelReadComplete()。整个过程还是比较清晰的，下面我们具体分析这两个方法。

1.创建NioSocketChannel：doReadMessages(List)。

2.设置并绑定NioSocketChannel：pipeline.fireChannelRead(NioSocketChannel)。

23.3.1 创建NioSocketChannel

NioMessageUnsafe.java

```
public void read() {  
  
    do {  
  
        // 1.创建 NioSocketChannel  
  
        int localRead = doReadMessages(readBuf);  
  
        if (localRead == 0) {  
  
            break;  
  
        }  
  
    } while (allocHandle.continueReading());  
  
    int size = readBuf.size();  
  
    // 2.设置并绑定 NioSocketChannel  
  
    // ...  
  
}
```

doReadMessages的方法体在NioServerSocketChannel类中，下面进入这个方法体来分析。

NioServerSocketChannel.java

```
protected int doReadMessages(List<Object> buf) throws Exception {  
  
    // 1. 创建 JDK 领域的 Channel  
  
    SocketChannel ch = javaChannel().accept();  
  
    // 2. 封装为 Netty 领域的 Channel  
  
    if (ch != null) {  
  
        buf.add(new NioSocketChannel(this, ch));  
  
        return 1;  
  
    }  
  
    return 0;  
  
}
```

这里的代码笔者也精简了一下。在这里，我们终于窥探到Netty调用JDK NIO的边界：javaChannel().accept()。由于Netty中Reactor线程第一步就扫描到有ACCEPT事件发生，因此，这里的accept方法是立即返回的，返回JDK底层NIO创建的一条JDK层面的Channel。

接下来，Netty将JDK的SocketChannel封装成自定义的NioSocketChannel，加入List，这样外层就可以遍历该List，做后续处理。

我们已经知道，服务端启动过程中会创建一个NioServerSocketChannel，而创建NioServerSocketChannel的过程中又会创建Netty的一系列核心组件，包括Pipeline、Unsafe等，那么，创建NioSocketChannel的时候是否也会创建这一系列组件呢？

带着这个疑问，我们分析NioSocketChannel的构造方法。

NioSocketChannel.java

```
public NioSocketChannel(Channel parent, SocketChannel socket) {  
  
    super(parent, socket);  
  
    config = new NioSocketChannelConfig(this, socket.socket());  
  
}
```

我们重点分析super(parent,socket), NioSocketChannel的父类为AbstractNioByteChannel。

AbstractNioByteChannel.java

```
protected AbstractNioByteChannel(Channel parent, SelectableChannel ch)  
{  
  
    super(parent, ch, SelectionKey.OP_READ);  
  
}
```

这里，我们看到JDK NIO里熟悉的影子SelectionKey.OP_READ，一般在原生的JDK NIO编程中，我们也会注册这样一个事件，表示对Channel的读事件感兴趣。

我们继续往上追踪，追踪到AbstractNioByteChannel的父类AbstractNioChannel。这里，相信大家应该已经了解了，NioServerSocketChannel最终的父类也是AbstractNioChannel。所以，创建NioSocketChannel的模板和创建NioServerSocketChannel保持一致。

```
protected AbstractNioChannel(Channel parent, SelectableChannel ch, int  
readInterestOp) {  
  
    super(parent);  
  
    this.ch = ch;  
  
    this.readInterestOp = readInterestOp;  
  
    try {  
  
        ch.configureBlocking(false);  

```

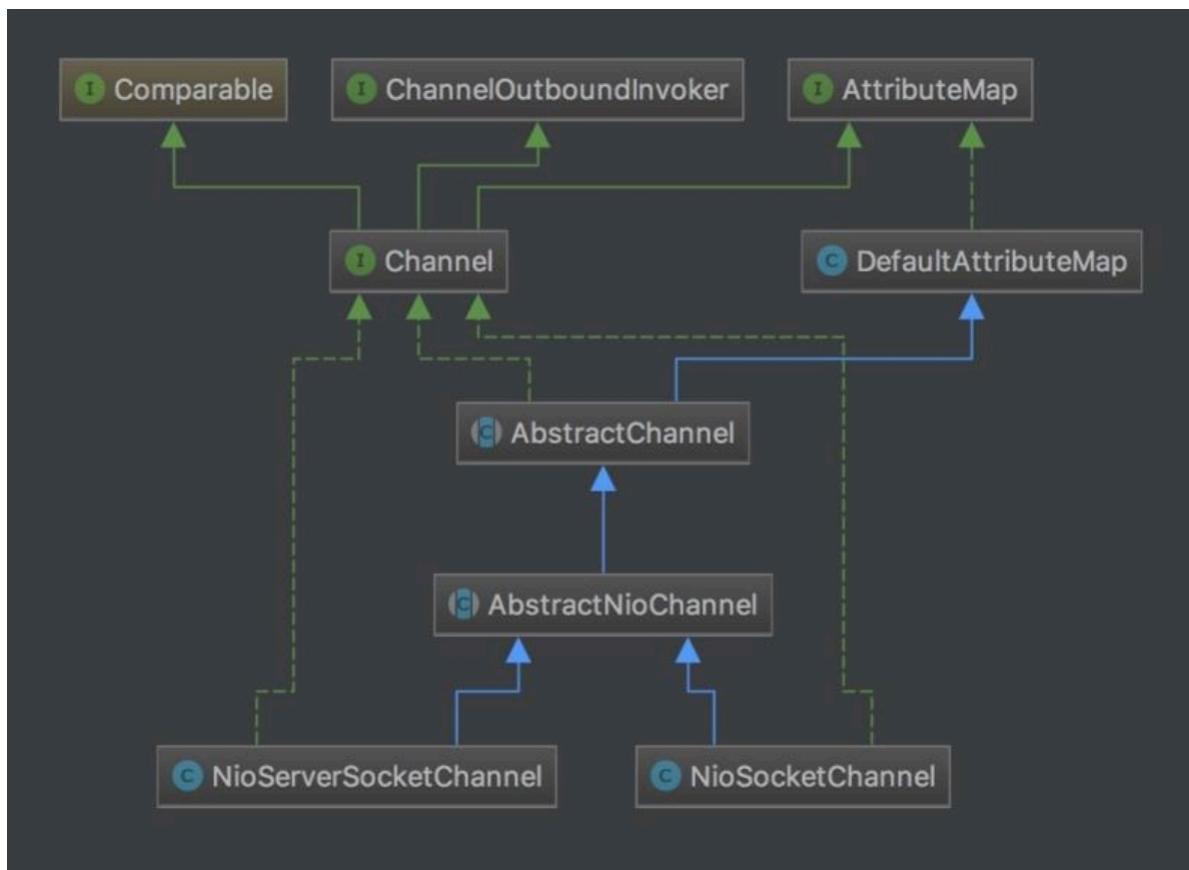
```
    } catch (IOException e) {  
  
        try {  
  
            ch.close();  
  
        } catch (IOException e2) {  
  
            if (logger.isWarnEnabled()) {  
  
                logger.warn(  
                    "Failed to close a partially initialized socket.", e2);  
  
            }  
  
        }  
  
        throw new ChannelException("Failed to enter non-blocking mode.", e);  
  
    }  
  
}
```

这里的readInterestOp表示该Channel关心的事件是SelectionKey.OP_READ，后续会将该事件注册到Selector，之后设置该通道为非阻塞模式。

AbstractNioChannel构造方法的第一行代码调用super(parent)，便是在AbstractChannel构造方法中创建一系列与该Channel绑定的组件。

```
protected AbstractChannel(Channel parent) {  
  
    this.parent = parent;  
  
    id = newId();  
  
    unsafe = newUnsafe();  
  
    pipeline = newChannelPipeline();  
  
}
```

分析到这里，是时候了解一下Netty中最常用的Channel的结构了，如下图所示。



这里的继承关系有所简化，当前，我们只需要了解这么多。

- 1.Channel继承Comparable表示Channel是一个可以比较的对象。
- 2.Channel继承AttributeMap表示Channel是可以绑定属性的对象，在用户代码中，我们经常使用channel.attr(...)来给Channel绑定属性，其实就是把属性设置到AttributeMap中。
- 3.ChannelOutboundInvoker是4.1.x版本新加的抽象，表示用户代码可以在Channel上进行哪些操作。
- 4.DefaultAttributeMap为AttributeMap的默认实现，后面的Channel继承了它，可以直接使用。
- 5.AbstractChannel用于实现Channel的大部分方法，其中我们最熟悉的就是在其构造方法中，创建一条Channel的基本组件，这里的Channel通常包括SocketChannel和ServerSocketChannel。

6. AbstractNioChannel基于AbstractChannel做了NIO相关的一些操作，保存JDK底层的SelectableChannel的引用，并且在构造方法中设置Channel为非阻塞。设置非阻塞这一点对于NIO编程是必不可少的。

7. 最后，就是两大Channel—NioServerSocketChannel和NioSocketChannel，分别对应着服务端接收新连接过程和新连接读写过程。

读到这，相信读者关于Channel的整体框架基本已经了解了一大半了。

我们继续之前的源码分析，在创建一条NioSocketChannel并放置在List容器里后，就开始for循环进行下一步操作。

23.3.2 设置并绑定NioSocketChannel

创建完NioSocketChannel之后，接下来要对NioSocketChannel做一些设置，并且需要将它绑定到一个执行的Reactor线程中。

NioMessageUnsafe.java

```
public void read() {  
    // 1. 创建 NioSocketChannel  
  
    // 2. 设置并绑定 NioSocketChannel  
  
    int size = readBuf.size();  
  
    for (int i = 0; i < size; i++) {  
  
        pipeline.fireChannelRead(readBuf.get(i));  
  
    }  
}
```

readBuf中承载着所有新建的连接，如果某个时刻，Netty轮询到多个连接，那么使用for循环就可以批量处理这些连接，即NioSocketChannel。

处理每一个NioSocketChannel，是通过调用NioServerSocketChannel的pipeline.fireChannelRead来执行的，在后面章节正式介绍Pipeline之前，笔者先简单介绍一下Pipeline组件。

在Netty的各种类型的Channel中，都会包含一个Pipeline。Pipeline的字面意思是管道，我们可以理解为一条流水线。流水线有起点，有结束，中间还有各种各样的流水线关卡。一件物品，在流水线起点开始处理，经过各个流水线关卡的加工，最终到流水线结束。

对应到Netty里，流水线的开始是HeadContext，流水线的结束是TailContext。HeadContext中调用Unsafe做具体的操作，TailContext中用于向用户抛出Pipeline中未处理异常及对未处理消息的警告。我们暂时先了解这么多，关于Pipeline的具体分析，我们放到后面的章节再详细探讨。

在服务端的启动过程中，Netty给服务端Channel自动添加了一个Pipeline处理器ServerBootstrap-Acceptor，并已经将用户代码中设置的一系列参数传入了ServerBootstrapAcceptor构造方法。接下来，我们来分析ServerBootstrapAcceptor。

ServerBootstrapAcceptor.java

```
private static class ServerBootstrapAcceptor extends  
ChannelInboundHandlerAdapter {  
  
    private final EventLoopGroup childGroup;  
  
    private final ChannelHandler childHandler;  
  
    private final Entry<ChannelOption<?>, Object> [] childOptions;  
  
    private final Entry<AttributeKey<?>, Object> [] childAttrs;  
  
    ServerBootstrapAcceptor(  
  
        EventLoopGroup childGroup, ChannelHandler childHandler,  
  
        Entry<ChannelOption<?>, Object> [] childOptions,  
        Entry<AttributeKey<?>,  
        Object> [] childAttrs) {  
  
        this.childGroup = childGroup;  
  
        this.childHandler = childHandler;  
  
        this.childOptions = childOptions;
```

```
this.childAttrs = childAttrs;

    }

// channelRead 方法在新连接接入时被调用

public void channelRead(ChannelHandlerContext ctx, Object msg) {

final Channel child = (Channel) msg;

// 1. 给新连接的 Channel 添加用户自定义的 Handler 处理器, 这里其实是一个特殊的

Handler: ChannelInitializer

child.pipeline().addLast(childHandler);

// 2. 设置ChannelOption, 主要和TCP连接一些底层参数及Netty自身对一个连接的参数有关

for (Entry<ChannelOption<?>, Object> e: childOptions) {

if (! child.config().setOption((ChannelOption<Object>) e.getKey(),
e.getValue()))

{

logger.warn("Unknown channel option: " + e);

}

}

// 3. 设置新连接 Channel 的属性

for (Entry<AttributeKey<?>, Object> e: childAttrs) {

child.attr((AttributeKey<Object>) e.getKey()).set(e.getValue());

}

}
```

```
// 4. 绑定 Reactor 线程  
  
childGroup.register(child);  
  
}
```

1.添加用户自定义Handler。pipeline.fireChannelRead(NioSocketChannel)最终调用这里的ServerBootstrapAcceptor的channelRead方法，而channelRead一上来就把这里的msg强制转换为Channel，为什么这里可以强制转换？读者可以思考一下。

拿到该Channel，也就是拿到了该Channel对应的Pipeline，这个Pipeline其实就是在第23.3.1节中调用AbstractChannel的构造方法时创建的。然后，将用户代码中的childHandler，添加到Pipeline中。这里的childHandler在用户代码中的体现如下。

```
ServerBootstrap b = new ServerBootstrap();  
  
b.group(bossGroup, workerGroup)  
  
.channel(NioServerSocketChannel.class)  
  
.childHandler(new ChannelInitializer<SocketChannel>() {  
  
    @Override  
  
    public void initChannel(SocketChannel ch) throws Exception {  
  
        ChannelPipeline p = ch.pipeline();  
  
        p.addLast(new EchoServerHandler());  
  
    }  
  
});
```

childHandler对应的就是上述用户代码中的ChannelInitializer。到了这里，NioSocketChannel中Pipeline对应的Handler为head->ChannelInitializer->tail。

2.设置ChannelOption。这里的ChannelOption也在用户代码中设置，最终传递到ServerBootstrapAcceptor。ChannelOption主要是和TCP底层参数相关的一些配置及Netty对一条连接的配置。

3.设置ChannelAttr。设置NioSocketChannel对应的ChannelAttr，ChannelAttr和ChannelOption一样，也在用户代码中设置，最终传递到ServerBootstrapAcceptor，一般情况下用不着ChannelAttr。

4.绑定Reactor线程。对于childGroup.register(child)，这里的childGroup就是我们在用户代码里创建的workerNioEventLoopGroup，我们进入NioEventLoopGroup的register方法，register首先调用next()方法获取一个EventLoop对象MultithreadEventLoopGroup。

MultithreadEventLoopGroup.java

```
public ChannelFuture register(Channel channel) {  
  
    return next().register(channel);  
  
}  
  
@Override  
  
public EventLoop next() {  
  
    return (EventLoop) super.next();  
  
}
```

调用其父类MultithreadEventExecutorGroup。

MultithreadEventExecutorGroup.java

```
@Override  
  
public EventExecutor next() {  
  
    return chooser.next();  
  
}
```

我们发现，MultithreadEventExecutorGroup中的next()方法调用了chooser对象的next()方法，而这个对象正是我们在第22.1.5节分析的EventExecutorChooser，它的作用是从NioEventLoopGroup中，选择一个NioEventLoop，所以，最终

childGroup.register(child)会调用NioEventLoop的register方法，由其父类SingleThreadEventLoop来实现。

SingleThreadEventLoop.java

```
@Override  
  
public ChannelFuture register(Channel channel) {  
  
    return register(new DefaultChannelPromise(channel, this));  
  
}
```

到这里，读者应该会比较眼熟了，这里和服务端启动流程中的注册Channel的模板一样，都由AbstractUnsafe来执行。下面我们来分析，对应新连接接入，这一套逻辑应该如何执行。

最终，register方法会调用如下方法。

AbstractUnsafe

```
private void register0(ChannelPromise promise) {  
  
    boolean firstRegistration = neverRegistered;  
  
    // 1. 注册 Selector  
  
    doRegister();  
  
    neverRegistered = false;  
  
    registered = true;  
  
    // 2. 配置自定义 Handler  
  
    pipeline.invokeHandlerAddedIfNeeded();  
  
    safeSetSuccess(promise);  
  
    // 3. 传播 ChannelRegistered 事件  
  
    pipeline.fireChannelRegistered();
```

```
// 4. 注册读事件

if (isActive()) {

if (firstRegistration) {

pipeline.fireChannelActive();

} else if (config().isAutoRead()) {

beginRead();

}

}

}

}
```

1.注册Selector。和服务端启动过程一样，先调用doRegister()进行真正的注册过程。

```
protected void doRegister() throws Exception {

boolean selected = false;

for (; ; ) {

try {

selectionKey = javaChannel().register(eventLoop().selector, 0,
this);

return;

} catch (CancelledKeyException e) {

if (! selected) {

eventLoop().selectNow();

selected = true;

} else {
```

`javaChannel().register`将`NioSocketChannel`绑定到Reactor线程的`Selector`上，这样，后续该`NioSocketChannel`所有的事件都由绑定的Reactor线程的`Selector`来轮询。

2. 配置自定义Handler。到目前为止，`NioSocketChannel`的Pipeline中有三个Handler：`head->ChannelInitializer->tail`。接下来，`invokeHandlerAddedIfNeeded`最终会调用`ChannelInitializer`的`handlerAdded`方法。

ChannelInitializer.java

```
public void handlerAdded(ChannelHandlerContext ctx) throws Exception {  
  
    if (ctx.channel().isRegistered()) {  
  
        initChannel(ctx);  
  
    }  
  
}  
  
}
```

handlerAdded方法调用initChannel方法之后，调用remove(ctx)将自身删除。

AbstractNioChannel.java

```
private boolean initChannel(ChannelHandlerContext ctx) throws  
Exception {  
  
    if (initMap.putIfAbsent(ctx, Boolean.TRUE) == null) {  
  
        try {  
  
            initChannel((C) ctx.channel());  
  
        } catch (Exception e) {  
            logger.error("initChannel error", e);  
        }  
    }  
}
```

```
        } catch (Throwable cause) {  
  
            exceptionCaught(ctx, cause);  
  
        } finally {  
  
            remove(ctx);  
  
        }  
  
    return true;  
  
}  
  
return false;  
  
}
```

这里的initChannel方法又是什么呢？让我们回到服务端启动代码，比如下面这段用户代码。

用户代码

```
ServerBootstrap b = new ServerBootstrap();  
  
b.group(bossGroup, workerGroup)  
  
.channel(NioServerSocketChannel.class)  
  
.option(ChannelOption.SO_BACKLOG, 100)  
  
.handler(new LoggingHandler(LogLevel.INFO))  
  
.childHandler(new ChannelInitializer<SocketChannel>() {  
  
    @Override  
  
    public void initChannel(SocketChannel ch) throws Exception {  
  
        ChannelPipeline p = ch.pipeline();  
  
        p.addLast(new LoggingHandler(LogLevel.INFO));  
    }  
});
```

```
p.addLast(new EchoServerHandler());  
}  
};
```

对照前面的分析，原来，最终initChannel会调用用户代码，而一般在用户代码里，我们会添加自定义的系列Handler。所以，整个过程其实就是给NioSocketChannel配置自定义Handler，NioSocketChannel中的Handler包括head->LoggingHandler->EchoServerHandler->tail。

3.传播ChannelRegistered事件。pipeline.fireChannelRegistered()其实没有干特别的事情，最终只是把连接注册事件往下传播，调用了每一个Handler的channelRegistered方法。

4.注册读事件。现在，我们还剩下这些代码没有分析。

AbstractNioChannel.java

```
private void register0(ChannelPromise promise) {  
  
    // 4. 注册读事件  
  
    if (isActive()) {  
  
        if (firstRegistration) {  
  
            pipeline.fireChannelActive();  
  
        } else if (config().isAutoRead()) {  
  
            beginRead();  
  
        }  
    }  
}
```

isActive()在连接已经建立的情况下返回true，所以进入方法块，即进入pipeline.fireChannelActive()。接下来的调用过程和服务端启动流程的分析过程一样，

最终都会调用如下代码。

AbstractNioChannel.java

```
@Override  
  
protected void doBeginRead() throws Exception {  
  
    final SelectionKey selectionKey = this.selectionKey;  
  
    final int interestOps = selectionKey.interestOps();  
  
    if ((interestOps & readInterestOp) == 0) {  
  
        selectionKey.interestOps(interestOps | readInterestOp);  
  
    }  
  
}
```

读者应该还记得前面分析register方法的时候，向Selector注册的事件代码是0，而readInterestOp对应的事件代码是SelectionKey.OP_READ。参考前文中创建NioSocketChannel的过程，稍加推理，就会知道，这里其实就是将SelectionKey.OP_READ事件注册到Selector，表示这条管道已经可以开始处理读事件。至此，新连接接入的流程就算结束了。

23.3.3 注册 Reactor 线程小结

当boss Reactor线程在检测到有ACCEPT事件之后，创建JDK底层的Channel，然后使用一个NioSocketChannel包装JDK底层的Channel，把用户设置的ChannelOption、ChannelAttr、ChannelHandler都设置到NioSocketChannel中。

接着，从worker Reactor线程组，也就是worker NioEventLoopGroup选择一个NioEventLoop，把NioSocketChannel包装的JDK的Channel当作key，自身当作attachment，注册到NioEventLoop对应的Selector。这样，后续有读写事件发生时，就可以直接获得attachment，也就是NioSocketChannel，来处理读写数据逻辑。

23.4 总结

最后，我们对本章内容进行总结。

- 1.boss Reactor线程轮询到有新连接进入。
- 2.通过封装JDK底层的Channel创建NioSocketChannel及一系列Netty核心组件。
- 3.通过chooser选择一个worker Reactor线程将该连接绑定上去。
- 4.注册读事件，开始新连接的读写。

第24章

解码原理解析

本章我们分析Netty中解码相关的逻辑是如何实现的。

24.1 粘包与拆包

在前面的内容中，我们已经了解了拆包的概念，我们再花点时间回顾一下。

24.1.1 为什么要粘包

首先你得了解一下TCP/IP协议，在用户数据量非常小的情况下，比如1字节，该TCP数据包的有效载荷非常低，传递100字节的数据，需要100次TCP传送、100次ACK，在应用及时性要求不高的情况下，将这100个有效数据拼接成一个数据包，就会缩短到一个TCP数据包，以及一个ACK，提高了有效载荷，也节省了带宽。

在非极端情况下，有可能两个数据包拼接成一个数据包，也有可能一个半的数据包拼接成一个数据包，也有可能两个半的数据包拼接成一个数据包。

24.1.2 为什么要拆包

拆包和粘包是相对的，一端粘了包，另外一端就需要将粘过的包拆开。举个例子，发送端将三个数据包粘成两个数据包发送到接收端，接收端就需要根据应用协议将两个数据包重新组装成三个数据包。

还有一种情况就是用户数据包超过了mss（最大报文长度），那么这个数据包在发送的时候必须拆分成几个数据包，接收端收到这些数据包之后需要将这些数据包粘合之后再拆开。

24.2 拆包的原理

在没有Netty的情况下，用户如果自己需要拆包，基本原理就是不断地从TCP缓冲区中读取数据，每次读取完都需要判断是否为一个完整的数据包。

1.如果当前读取的数据不足以拼接成一个完整的业务数据包，那就保留该数据，继续从TCP缓冲区中读取，直到得到一个完整的数据包。

2.如果当前读到的数据加上已经读取的数据足够拼接成一个数据包，那就将已经读取的数据拼接上本次读取的数据，构成一个完整的业务数据包传递到业务逻辑，多余的数据仍然保留，以便和下次读到的数据尝试拼接。

24.3 Netty中拆包的基类

Netty中的拆包原理也如上，内部会有一个累加器，每次读取到数据都会不断累加，然后尝试对累加的数据进行拆包，拆成一个完整的业务数据包，这个基类叫作ByteToMessageDecoder，下面我们详细分析这个类。

ByteToMessageDecoder中定义了两个累加器。

```
public static final Cumulator MERGE_CUMULATOR = ...;
```

```
public static final Cumulator COMPOSITE_CUMULATOR = ...;
```

在默认情况下，会使用MERGE_CUMULATOR。

```
private Cumulator cumulator = MERGE_CUMULATOR;
```

MERGE_CUMULATOR的原理是每次都将读取到的数据通过内存拷贝的方式，拼接到一个大的字节容器中，这个字节容器在ByteToMessageDecoder中被叫作cumulation。

```
ByteBuf cumulation;
```

下面我们看一下MERGE_CUMULATOR是如何将新读取的数据累加到字节容器里的。

ByteToMessageDecoder.java

```
public ByteBuf cumulate(ByteBufAllocator alloc, ByteBuf cumulation,
    ByteBuf in)  {

    ByteBuf buffer;

    if (cumulation.writerIndex() > cumulation.maxCapacity() -
        in.readableBytes() || cumulation.refCnt() > 1)  {

        buffer = expandCumulation(alloc, cumulation, in.readableBytes());

    } else {
```

```
        buffer = cumulation;

    }

    buffer.writeBytes(in);

    in.release();

    return buffer;

}
```

Netty中ByteBuf的抽象，使得累加非常简单，通过一个简单的API调用
buffer.writeBytes(in);便可以将新数据累加到字节容器中，为了防止字节容器容量不
够，在累加之前还进行了扩容处理。

```
static ByteBuf expandCumulation(ByteBufAllocator alloc, ByteBuf
cumulation, int readable) {

    ByteBuf oldCumulation = cumulation;

    cumulation = alloc.buffer(oldCumulation.readableBytes() + readable);

    cumulation.writeBytes(oldCumulation);

    oldCumulation.release();

    return cumulation;

}
```

扩容也是一个内存拷贝操作，新增的大小即新读取数据的大小。

24.4 拆包抽象

累加器原理清楚之后，我们回到主流程，目光集中在channelRead方法上。
channelRead方法是每次从TCP缓冲区读到数据都会调用的方法，触发点在
AbstractNioByteChannel的read方法中，里面有一个while循环不断读取数据，读取到
一次就触发一次channelRead方法。

ByteToMessageDecoder.java

```
@Override

public void channelRead(ChannelHandlerContext ctx, Object msg) throws
Exception {
    // 1. 累加数据

    // 2. 将累加的数据传递给业务进行拆包

    // 3. 清理字节容器

    // 4. 将业务数据包传递给业务解码器处理
}
```

方法体不长不短，可以分为以下几个逻辑步骤。

- 1.累加数据。
- 2.将累加的数据传递给业务进行拆包。
- 3.清理字节容器。
- 4.将业务数据包传递给业务解码器处理。

24.4.1 累加数据

如果当前累加器中没有数据，就直接跳过内存拷贝，将字节容器的指针指向新读取的数据，否则，调用累加器累加数据至字节容器。

ByteToMessageDecoder.java

```
@Override

public void channelRead(ChannelHandlerContext ctx, Object msg) throws
Exception {
    // 1. 累加数据

    ByteBuf data = (ByteBuf) msg;
    first = cumulation == null;
```

```
if (first) {  
  
    cumulation = data;  
  
} else {  
  
    cumulation = cumulator.cumulate(ctx.alloc(), cumulation, data);  
  
}  
  
// 2. 将累加的数据传递给业务进行拆包  
  
// 3. 清理字节容器  
  
// 4. 将业务数据包传递给业务解码器处理  
  
}
```

24.4.2 将累加的数据传递给业务进行拆包

到这一步，字节容器里的数据已经是目前未拆包部分的所有数据了。

ByteToMessageDecoder.java

```
@Override  
  
public void channelRead(ChannelHandlerContext ctx, Object msg) throws  
Exception {  
  
    // 1. 累加数据  
  
    // 2. 将累加的数据传递给业务进行拆包  
  
    CodecOutputList out = CodecOutputList.newInstance();  
  
    callDecode(ctx, cumulation, out);  
  
    // 3. 清理字节容器  
  
    // 4. 将业务数据包传递给业务解码器处理  
  
}
```

callDecode尝试将字节容器的数据拆分成业务数据包，塞到业务数据容器out中。

ByteToMessageDecoder.java

```
protected void callDecode(ChannelHandlerContext ctx, ByteBuf in,
List<Object> out) {

    while (in.isReadable()) {

        // 记录一下字节容器中有多少字节待拆

        int oldInputLength = in.readableBytes();

        decode(ctx, in, out);

        if (out.size() == 0) {

            // 拆包器未读取任何数据

            if (oldInputLength == in.readableBytes()) {

                break;

            } else {

                // 拆包器已读取部分数据，还需要继续

                continue;

            }

        }

        if (oldInputLength == in.readableBytes()) {

            throw new DecoderException(

StringUtil.simpleClassName(getClass()) +

".decode() did not read anything but decoded a message.");

        }

    }

}
```

```
if (isSingleDecode()) {  
  
    break;  
  
}  
  
}  
  
}
```

笔者将原始代码做了一些精简，在解码之前，先记录一下字节容器中有多少字节待拆，然后调用抽象函数decode进行拆包。

```
protected abstract void decode(ChannelHandlerContext ctx, ByteBuf in,  
List<Object> out) throws Exception;
```

Netty中对各种用户协议的支持就体现在这个抽象函数中，传进去的是当前读取到的未被消费的所有数据，以及业务协议包容器，所有的拆包器最终都实现了该抽象方法。

业务拆包完成之后，如果发现并没有拆到一个完整的数据包，这个时候又分成两种情况。

1.一种是拆包器什么数据也没读取，可能数据还不够业务拆包器处理，通过break关键字跳出循环。

2.另一种是拆包器已读取部分数据，说明解码器仍然在工作，继续解码。

业务拆包完成之后，如果发现已经解到了数据包，但是发现并没有读取任何数据，这个时候就会抛出一个Runtime异常DecoderException，告诉你什么数据都没读取到，却解析出一个业务数据包，这是有问题的。

24.4.3 清理字节容器

业务拆包完成之后，只是从字节容器中取走了数据，但是这部分空间对于字节容器来说依然被保留着，而字节容器每次累加字节数据的时候都是将字节数据追加到尾部，如果不清理字节容器，那么时间一长就会出现OOM问题。

在正常情况下，其实每次读取完数据，Netty都会在下面这个方法中将字节容器清理，只不过，当发送端发送数据过快时，channelReadComplete可能会很久才被调用一次。

```

public void channelReadComplete(ChannelHandlerContext ctx) throws
Exception {
    numReads = 0;

    discardSomeReadBytes();

    if (decodeWasNull) {
        decodeWasNull = false;

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

ctx.fireChannelReadComplete();

}

```

这里顺带插一句，如果一次数据读取完毕之后（可能接收端一边收，发送端一边发，这里的读取完毕指接收端在某个时间不再接收数据），发现仍然没有拆到一个完整的用户数据包，即使该Channel的设置为非自动读取，也会触发一次读取操作ctx.read()，该操作会重新向Selector注册op_read事件，以便下一次能在读到数据后拼接成一个完整的数据包。

所以为了防止发送端发送数据过快，Netty会在每次读取到一次数据，业务拆包之后，对字节容器做清理。清理字节容器部分的代码如下。

ByteToMessageDecoder.java

```

@Override

public void channelRead(ChannelHandlerContext ctx, Object msg) throws
Exception {
    // 1. 累加数据

```

```
// 2. 将累加的数据传递给业务进行拆包
```

// 3. 清理字节容器

```
if (cumulation != null && ! cumulation.isReadable()) {  
  
    numReads = 0;  
  
    cumulation.release();  
  
    cumulation = null;  
  
} else if (++ numReads >= discardAfterReads) {  
  
    numReads = 0;  
  
    discardSomeReadBytes();  
  
}
```

```
// 4. 将业务数据包传递给业务解码器处理
```

```
}
```

如果字节容器当前已无数据可读取，则直接销毁字节容器，并且标注当前字节容器一次数据也没读取。

如果连续读取16次（discardAfterReads的默认值），字节容器中仍然有未被业务拆包器读取的数据，那么就做一次压缩，把有效数据段整体移到容器首部。

在discardSomeReadBytes之前，字节累加器中的数据分布如下。

| ----- | ----- | ----- |
|-------|--------|---------------------|
| | readed | unreaded writable |
| ----- | ----- | ----- |

在discardSomeReadBytes之后，字节容器中的数据分布如下。

| unreaded | writable |
|----------|----------|
| | |

这样字节容器又可以承载更多数据了。

24.4.4 将业务数据包传递给业务解码器处理

以上三个步骤完成之后，就可以将拆成的数据包丢到业务解码器中处理了，代码如下。

ByteToMessageDecoder.java

```
public void channelRead(ChannelHandlerContext ctx, Object msg) throws
Exception {
    // 1. 累加数据
    // 2. 将累加的数据传递给业务进行拆包
    // 3. 清理字节容器
    // 4. 将业务数据包传递给业务解码器处理
    int size = out.size();
    decodeWasNull = !out.insertSinceRecycled();
    fireChannelRead(ctx, out, size);
    out.recycle();
}
```

期间用一个成员变量decodeWasNull来标识本次读取数据是否拆到一个业务数据包，然后调用fireChannelRead将拆到的业务数据包都传递到后续的Handler。

```
static void fireChannelRead(ChannelHandlerContext ctx, CodecOutputList
msgs, int numElements) {
    for (int i = 0; i < numElements; i++) {
```

```
    ctx.fireChannelRead(msgs.getUnsafe(i));  
  
    }  
  
}
```

这样，就可以把一个个完整的业务数据包传递到后续的业务解码器进行解码，随后处理业务逻辑。

24.5 行拆包器

下面以一个具体的例子来看看Netty自带的拆包器是如何拆包的。

这个类叫作LineBasedFrameDecoder，是基于行分隔符的拆包器，它可以同时处理\n和\r\n两种类型的行分隔符，核心方法都在继承的decode方法中。

LineBasedFrameDecoder.java

```
protected final void decode(ChannelHandlerContext ctx, ByteBuf in,  
List<Object> out) throws Exception {  
  
    Object decoded = decode(ctx, in);  
  
    if (decoded != null) {  
  
        out.add(decoded);  
  
    }  
  
}
```

Netty中自带的拆包器都是基于如上这种模板的，其实可以加一层，把这层模板抽取出来，不知道为什么Netty没有这么做。我们接着跟进去，代码比较长，我们还是分模块来剖析。

24.5.1 找到换行符的位置

LineBasedFrameDecoder.java

```
final int eol = findEndOfLine(buffer);
```

```
private static int findEndOfLine(final ByteBuf buffer) {  
  
    int i = buffer.forEachByte(ByteProcessor.FIND_LF);  
  
    if (i > 0 && buffer.getByte(i - 1) == '\r') {  
  
        i--;  
  
    }  
  
    return i;  
}  
  
ByteProcessor FIND_LF = new IndexOfProcessor((byte) '\n');
```

for循环遍历，找到第一个\n的位置，如果\n前面的字符为\r，那么返回\r的位置。

24.5.2 非discarding模式的处理

接下来，Netty会判断，当前拆包是否处于discarding模式，用一个成员变量来标识。

```
private boolean discarding;
```

第一次拆包不在discarding模式中（后面的分支会讲何为非discarding模式），于是进入以下环节。

24.5.2.1 非discarding模式下找到行分隔符的处理

```
// 1.计算分隔符和包长度  
  
final ByteBuf frame;  
  
final int length = eol - buffer.readerIndex();  
  
final int delimLength = buffer.getByte(eol) == '\r'? 2 : 1;  
  
// 丢弃异常数据  
  
if (length > maxLength) {  
  
    buffer.readerIndex(eol + delimLength);
```

```

    fail(ctx, length);

    return null;

}

// 取包的时候是否包括分隔符

if (stripDelimiter) {

    frame = buffer.readRetainedSlice(length);

    buffer.skipBytes(delimLength);

} else {

    frame = buffer.readRetainedSlice(length + delimLength);

}

return frame;

```

- 1.新建一个帧，计算当前包的长度和分隔符的长度（因为有两种分隔符）。
- 2.判断一下需要拆包的长度是否大于该拆包器允许的最大长度（maxLength），这个参数在构造函数中被传递进来，如果超出允许的最大长度，则将这段数据抛弃，返回null。
- 3.将一个完整的数据包取出，如果构造本拆包器的时候，指定stripDelimiter为false，则解析出来的数据包包含分隔符。默认是不包含分隔符的。

24.5.2.2 非discarding模式下未找到分隔符的处理

没有找到对应的行分隔符，说明字节容器没有足够的数据拼接成一个完整的业务数据包，则进入如下流程处理。

```

final int length = buffer.readableBytes();

if (length > maxLength) {

    discardedBytes = length;

```

```
buffer.readerIndex(buffer.writerIndex());  
  
discarding = true;  
  
if (failFast) {  
  
    fail(ctx, "over " + discardedBytes);  
  
}  
  
}  
  
return null;
```

首先取得当前字节容器的可读字节数，接着判断一下是否已经超过允许的最大长度，如果没有超过，则直接返回null，字节容器中的数据没有任何改变，否则就需要进入discarding模式。

使用一个成员变量discardedBytes来表示已经丢弃了多少数据，然后将字节容器的读指针移到写指针，意味着丢弃这一部分数据。设置成员变量discarding为true，表示当前处于discarding模式。如果设置了failFast，那么直接抛出异常，默认情况下failFast为false，即安静地丢弃数据。

24.5.3 discarding模式的处理

如果解包的时候处在discarding模式，也会有两种情况发生。

24.5.3.1 discarding模式下找到行分隔符的处理

在discarding模式下，如果找到分隔符，那么可以将分隔符之前的数据都丢弃。

```
final int length = discardedBytes + eol - buffer.readerIndex();  
  
final int delimLength = buffer.getByte(eol) == '\r'? 2 : 1;  
  
buffer.readerIndex(eol + delimLength);  
  
discardedBytes = 0;  
  
discarding = false;  
  
if (! failFast) {
```

```
fail(ctx, length);

    }
```

计算出分隔符的长度之后，直接把分隔符之前的数据全部丢弃，当然丢弃的字符也包括分隔符，经过这么一次丢弃，后面就有可能是正常的数据包，下一次解包的时候就会进入正常的解包流程。

24.5.3.2 discarding模式下未找到行分隔符的处理

这种情况比较简单，因为当前还处在discarding模式，没有找到行分隔符意味着当前一个完整的数据包还没丢弃完，当前读取的数据是丢弃的一部分，所以直接丢弃。

```
discardedBytes += buffer.readableBytes();
```

```
buffer.readerIndex(buffer.writerIndex());
```

24.6 特定分隔符拆包

这个类叫作DelimiterBasedFrameDecoder，可以传递给它一个分隔符列表，数据包会按照分隔符列表进行拆分，读者可以完全根据行拆包器的思路去分析这个DelimiterBasedFrameDecoder，这里不再赘述。

24.7 LengthFieldBasedFrameDecoder进阶用法

之所以Netty的拆包能如此强大，就是因为Netty将具体如何拆包抽象出一个decode方法，不同的拆包器实现不同的decode方法，就能实现不同协议的拆包。

接下来要介绍的就是最常见的通用拆包器LengthFieldBasedFrameDecoder，下面我们将深入介绍一下其用法及原理。

24.7.1 基于长度的拆包

如下图所示，这类数据包协议比较常见，前面几字节表示数据包的长度（不包括长度域），后面是具体的数据。拆完之后，数据包是一个完整的带有长度域的数据包（之后即可传递到应用层解码器进行解码），创建一个如下方式的LengthFieldBasedFrameDecoder即可实现这类协议。

| BEFORE DECODE (14 bytes) | AFTER DECODE (14 bytes) |
|---|--|
| +-----+-----+ Length Actual Content -----> Length Actual Content 0x000C "HELLO, WORLD" +-----+-----+ +-----+-----+ | +-----+-----+ Length Actual Content 0x000C "HELLO, WORLD" +-----+-----+ |

```
new LengthFieldBasedFrameDecoder(Integer.MAX, 0, 4);
```

// 对应的构造方法如下

```
public LengthFieldBasedFrameDecoder(  
    int maxFrameLength,  
    int lengthFieldOffset, int lengthFieldLength) {  
  
    // ...  
}
```

其中，

- 1.第一个参数是maxFrameLength，表示包的最大长度，超出包的最大长度，Netty将做一些特殊处理，后面会讲到。
- 2.第二个参数指长度域的偏移量lengthFieldOffset，这里是0，表示无偏移。
- 3.第三个参数指长度域长度lengthFieldLength，这里是4，表示长度域的长度为4。

24.7.2 基于长度的截断拆包

如果我们的应用层解码器不需要使用长度字段，那么我们希望Netty拆完包之后，是下图所示的样子。

| BEFORE DECODE (14 bytes) | AFTER DECODE (12 bytes) |
|--|--|
| +-----+-----+ Length Actual Content -----> Actual Content 0x000C "HELLO, WORLD" +-----+-----+ +-----+-----+ | +-----+-----+ Actual Content "HELLO, WORLD" +-----+-----+ |

长度域被截掉，我们只需要指定另外一个参数就可以实现，这个参数叫作 initialBytesToStrip，表示Netty拿到一个完整的数据包之后、向业务解码器传递之前，应该跳过多少字节。

```
new LengthFieldBasedFrameDecoder(Integer.MAX, 0, 4, 0, 4);
```

前面三个参数的含义和上文相同，第四个参数我们后面再讲，而第五个参数就是 initialBytesToStrip，这里为4，表示获取完一个完整的数据包之后，忽略前面的4字节，应用解码器拿到的就是不带长度域的数据包。

24.7.3 基于偏移长度的拆包

下面这种方式的二进制协议是更为普遍的，前面几个固定字节表示协议头，通常包含一些magicNumber、protocol version之类的meta信息，紧跟着后面的是一个长度域，表示包体有多少字节的数据，如下图所示。

| BEFORE DECODE (17 bytes) | | | AFTER DECODE (17 bytes) | | |
|--------------------------|----------|----------------|-------------------------|----------|----------------|
| Header 1 | Length | Actual Content | Header 1 | Length | Actual Content |
| 0xCAFE | 0x00000C | "HELLO, WORLD" | 0xCAFE | 0x00000C | "HELLO, WORLD" |

只需要基于第一种情况，调整第二个参数即可实现。

```
new LengthFieldBasedFrameDecoder(Integer.MAX, 4, 4);
```

lengthFieldOffset是4，表示跳过4字节之后才是长度域。

24.7.4 基于可调整长度的拆包

有时候，二进制协议可能会设计成下图所示的方式。

| BEFORE DECODE (17 bytes) | | | AFTER DECODE (17 bytes) | | |
|--------------------------|----------|----------------|-------------------------|----------|----------------|
| Length | Header 1 | Actual Content | Length | Header 1 | Actual Content |
| 0x00000C | 0xCAFE | "HELLO, WORLD" | 0x00000C | 0xCAFE | "HELLO, WORLD" |

即长度域在前，header在后，这种情况又是如何来调整参数达到我们想要的拆包效果的呢？

1. 长度域在数据包最前面表示无偏移，lengthFieldOffset为0。

2. 长度域的长度为3，即lengthFieldLength为3。

3. 长度域表示的包体的长度不包含header的长度，这里有另外一个参数，叫作lengthAdjustment，包体长度调整的大小，长度域的数值表示的长度加上这个修正值表示的就是带header的包，这里是12+2，表示header和包体共占14字节。

最后，代码实现如下。

```
new LengthFieldBasedFrameDecoder(Integer.MAX, 0, 3, 2, 0);
```

24.7.5 基于偏移可调整长度的截断拆包

更复杂一点的二进制协议带有两个header，如下图所示。

| BEFORE DECODE (16 bytes) | AFTER DECODE (13 bytes) |
|---|-------------------------|
| HDR1 Length HDR2 Actual Content -----> HDR2 Actual Content 0xCA 0x000C 0xFE "HELLO, WORLD" 0xFE "HELLO, WORLD" | |

拆完之后，丢弃HDR1，丢弃长度域，只剩下第二个header和有效包体。在这种协议中，一般HDR1可以表示magicNumber，表示应用只接收以该magicNumber开头的二进制数据，RPC里用得比较多。

我们仍然可以通过设置Netty的参数实现。

1. 长度域偏移为1，那么lengthFieldOffset为1。

2. 长度域长度为2，那么lengthFieldLength为2。

3. 长度域表示的包体的长度略过了HDR2，但是拆包的时候HDR2也被Netty当作包体的一部分来拆，HDR2的长度为1，那么lengthAdjustment为1。

4. 拆完之后，截掉了前面的3字节，那么initialBytesToStrip为3。

最后，代码实现如下。

```
new LengthFieldBasedFrameDecoder(Integer.MAX, 1, 2, 1, 3);
```

24.7.6 基于偏移可调整变异长度的截断拆包

前面的所有长度域表示的都是不带header的包体的长度，如果让长度域表示的是整个数据包的长度，比如下图所示的这种情况。

| BEFORE DECODE (16 bytes) | AFTER DECODE (13 bytes) |
|---|-------------------------|
| HDR1 Length HDR2 Actual Content -----> HDR2 Actual Content 0xCA 0x0010 0xFE "HELLO, WORLD" 0xFE "HELLO, WORLD" | |

其中长度域字段的值为16，其字段长度为2，HDR1的长度为1，HDR2的长度为1，包体的长度为12， $1+1+2+12=16$ ，又该如何设置参数呢？

这里除了长度域表示的含义和上一种情况不一样，其他都相同，因为Netty并不了解业务情况，你需要告诉Netty的是，长度域后面再跟多少字节就可以形成一个完整的数据包，这里显然是13字节，而长度域的值为16，因此减掉3才是真实的拆包所需要的长度，lengthAdjustment为-3。

以上6种情况是Netty源码里自带的6种典型的二进制协议，相信已经囊括了90%以上的场景。如果你的协议是基于长度的，那么可以考虑不用字节来实现，而是直接拿来用，或者继承它，做些简单的修改即可。

如此强大的拆包器其实现也是非常优雅的，下面我们来看Netty是如何实现的。

24.8 LengthFieldBasedFrameDecoder源码剖析

24.8.1 构造函数

关于LengthFieldBasedFrameDecoder的构造函数，我们只需要看一个就够了。

LengthFieldBasedFrameDecoder.java

```
public LengthFieldBasedFrameDecoder(  
    ByteOrder byteOrder, int maxFrameLength, int lengthFieldOffset, int  
    lengthFieldLength,  
    int lengthAdjustment, int initialBytesToStrip, boolean failFast) {  
  
    // 省略参数校验部分  
  
    this.byteOrder = byteOrder;
```

```
this.maxFrameLength = maxFrameLength;  
  
this.lengthFieldOffset = lengthFieldOffset;  
  
this.lengthFieldLength = lengthFieldLength;  
  
this.lengthAdjustment = lengthAdjustment;  
  
lengthFieldEndOffset = lengthFieldOffset + lengthFieldLength;  
  
this.initialBytesToStrip = initialBytesToStrip;  
  
this.failFast = failFast;  
  
}
```

构造函数做的事很简单，只是把传入的参数简单地保存在field里，这里的大多数field在前面已经阐述过，剩下的几个补充说明如下。

1.byteOrder表示字节流表示的数据是大端还是小端，用于长度域的读取。

2.lengthFieldEndOffset表示紧跟长度域字段后面的第一字节在整个数据包中的偏移量。

3.failFast，如果为true，则表示读取到长度域，它的值超过maxFrameLength，就抛出一个TooLongFrameException，而为false则表示只有当真正读取完长度域的值表示的字节之后，才会抛出TooLongFrameException。默认情况下设置为true，建议不要修改，否则可能会造成内存溢出。

24.8.2 实现拆包抽象

通过前面内容的介绍，我们已经知道，具体的拆包协议只需要实现以下代码即可。

void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out) 其中，in表示到目前为止还未拆的数据，拆完之后的包添加到out这个list中，即可实现包向下传递。

第一层实现比较简单，代码如下。

LengthFieldBasedFrameDecoder.java

```

@Override

protected final void decode(ChannelHandlerContext ctx, ByteBuf in,
List<Object> out) throws Exception {
    Object decoded = decode(ctx, in);

    if (decoded != null) {
        out.add(decoded);
    }
}

```

重载的protected函数decode做真正的拆包动作，下面详细分析一下这个重量级函数。

1. 获取待拆包的大小。

LengthFieldBasedFrameDecoder.java

```

// 如果当前可读字节还未达到长度域的偏移，说明肯定是读不到长度域的，则直接不读

if (in.readableBytes() < lengthFieldEndOffset) {
    return null;
}

// 拿到长度域的实际字节偏移

int actualLengthFieldOffset = in.readerIndex() + lengthFieldOffset;

// 拿到实际的未调整过的包长度

long frameLength = getUnadjustedFrameLength(in,
actualLengthFieldOffset, lengthFieldLength, byteOrder);

// 如果拿到的长度为负数，则直接跳过长度域并抛出异常

if (frameLength < 0) {

```

```
in.skipBytes(lengthFieldEndOffset);

throw new CorruptedFrameException(
    "negative pre-adjustment length field: " + frameLength);

}

// 调整包的长度，后面统一做拆分

frameLength += lengthAdjustment + lengthFieldEndOffset;
```

上面这段内容有一个扩展点getUnadjustedFrameLength，如果长度域代表的值表达的含义不是正常的int、short等基本类型，则可以重写这个函数。

LengthFieldBasedFrameDecoder.java

```
protected long getUnadjustedFrameLength(ByteBuf buf, int offset, int
length, ByteOrder order) {

    buf = buf.order(order);

    long frameLength;

    switch (length) {

        case 1:

            frameLength = buf.getUnsignedByte(offset);

            break;

        case 2:

            frameLength = buf.getUnsignedShort(offset);

            break;

        case 3:

            frameLength = buf.getUnsignedMedium(offset);
```

```

break;

case 4:

frameLength = buf.getInt(offset);

break;

case 8:

frameLength = buf.getLong(offset);

break;

default:

throw new DecoderException(
    "unsupported lengthFieldLength: " + lengthFieldLength + "
(expected:
1, 2, 3, 4, or 8)");
}

return frameLength;
}

```

比如，有的“奇葩”的长度域虽然是4字节，比如0x1234，但是它的含义是十进制的，即长度就是十进制的1234，那么覆盖这个函数即可实现“奇葩”长度域拆包。

2. 长度校验。

LengthFieldBasedFrameDecoder.java

```

// 整个数据包的长度还没有长度域长，直接抛出异常

if (frameLength < lengthFieldEndOffset) {
    in.skipBytes(lengthFieldEndOffset);
}

```

```
throw new CorruptedFrameException(
    "Adjusted frame length (" + frameLength + ") is less " +
    "than lengthFieldEndOffset: " + lengthFieldEndOffset);
}

// 数据包长度超出最大包长度，进入discarding模式

if (frameLength > maxFrameLength) {
    long discard = frameLength - in.readableBytes();
    tooLongFrameLength = frameLength;
    if (discard < 0) {
        // 当前可读字节已达到frameLength，直接跳过frameLength字节，丢弃之后，后面有可能就
        是一个合法的数据包
        in.skipBytes((int) frameLength);
    } else {
        // 当前可读字节未达到frameLength，说明后面未读到的字节也需要丢弃，进入
        discarding模
        式，先把当前累积的字节全部丢弃
        discardingTooLongFrame = true;
        // bytesToDiscard表示还需要丢弃多少字节
        bytesToDiscard = discard;
        in.skipBytes(in.readableBytes());
    }
}
```

```
failIfNecessary(true);

return null;

}
```

最后，调用failIfNecessary判断是否需要抛出异常。

LengthFieldBasedFrameDecoder.java

```
private void failIfNecessary(boolean firstDetectionOfTooLongFrame) {

    // 不需要再丢弃后面的未读字节，就开始重置丢弃状态

    if (bytesToDiscard == 0) {

        long tooLongFrameLength = this.tooLongFrameLength;

        this.tooLongFrameLength = 0;

        discardingTooLongFrame = false;

        // 如果没有设置快速失败，或者设置了快速失败并且是第一次检测到大包错误，抛出异常，让

    }
}
```

Handler去处理

```
if (! failFast ||

    failFast && firstDetectionOfTooLongFrame) {

    fail(tooLongFrameLength);

}

} else {

    // 如果设置了快速失败，并且是第一次检测到打包错误，则抛出异常，让Handler去处理

    if (failFast && firstDetectionOfTooLongFrame) {
```

```
    fail(tooLongFrameLength);

    }

}

}
```

前面我们可以知道failFast默认为true，而这里firstDetectionOfTooLongFrame为true，所以，第一次检测到大包肯定会抛出异常。

下面是抛出异常的代码。

LengthFieldBasedFrameDecoder.java

```
private void fail(long frameLength) {

    if (frameLength > 0) {

        throw new TooLongFrameException(
            "Adjusted frame length exceeds " + maxFrameLength +
            ": " + frameLength + " - discarded");

    } else {

        throw new TooLongFrameException(
            "Adjusted frame length exceeds " + maxFrameLength +
            " - discarding");

    }
}
```

3.discard模式的处理。如果读者是一边对着源码，一边阅读本节内容，就会发现LengthFieldBasedFrameDecoder的decoder函数的入口处还有一段代码在我们前面的分析中被省略掉了。放到这一节中的目的是承接上一节，更加容易读懂discarding模式的处理。

LengthFieldBasedFrameDecoder.java

```
if (discardingTooLongFrame) {  
  
    long bytesToDiscard = this.bytesToDiscard;  
  
    int localBytesToDiscard = (int) Math.min(bytesToDiscard,  
in.readableBytes());  
  
    in.skipBytes(localBytesToDiscard);  
  
    bytesToDiscard -= localBytesToDiscard;  
  
    this.bytesToDiscard = bytesToDiscard;  
  
    failIfNecessary(false);  
  
}
```

如上代码所示，如果当前处在discarding模式，先计算需要丢弃多少字节，取当前还需可丢弃字节和可读字节的最小值。丢弃掉后，进入failIfNecessary，对照这个函数看，默认情况下是不会继续抛出异常的；而如果设置了failFast为false，那么等丢弃完后，才会抛出异常，读者可自行分析。

4.跳过指定字节长度。discarding模式的处理及长度的校验都通过之后，进入跳过指定字节长度这个环节。

LengthFieldBasedFrameDecoder.java

```
int frameLengthInt = (int) frameLength;  
  
if (in.readableBytes() < frameLengthInt) {  
  
    return null;  
  
}  
  
if (initialBytesToStrip > frameLengthInt) {  
  
    in.skipBytes(frameLengthInt);  
  
    throw new CorruptedFrameException(  
}
```

```
"Adjusted frame length (" + frameLength + ") is less " +  
"than initialBytesToStrip: " + initialBytesToStrip);  
}  
  
in.skipBytes(initialBytesToStrip);
```

先验证当前是否已经读到足够的字节，如果读到了，则在下一步抽取一个完整的数据包之前，需要根据initialBytesToStrip的设置来跳过某些字节（见本节开篇）。当然，跳过的字节不能大于数据包的长度，否则就抛出CorruptedFrameException的异常。

5. 抽取frame。

LengthFieldBasedFrameDecoder.java

```
int readerIndex = in.readerIndex();  
  
int actualFrameLength = frameLengthInt - initialBytesToStrip;  
  
ByteBuf frame = extractFrame(ctx, in, readerIndex, actualFrameLength);  
  
in.readerIndex(readerIndex + actualFrameLength);  
  
return frame;
```

到了最后，抽取数据包其实就很简单了，拿到当前累积数据的读指针，然后拿到待抽取数据包的实际长度进行抽取，抽取之后，移动读指针。

LengthFieldBasedFrameDecoder.java

```
protected ByteBuf extractFrame(ChannelHandlerContext ctx, ByteBuf  
buffer, int index, int length) {  
  
    return buffer.retainSlice(index, length);  
}
```

抽取的过程就是简单地调用了一下ByteBuf的retainSliceapi，该API无内存拷贝开销。

从真正抽取数据包来看，传入的参数为int类型的，所以，可以判断，在自定义协议中，如果你的长度域是8字节的，那么前面4字节基本是没有用的。

24.9 总结

- 1.Netty中的拆包过程其实和你自己去拆包的过程一样，只不过它将拆包过程中逻辑比较独立的部分抽象出来变成几个不同层次的类，方便各种协议的扩展。我们平时在写代码过程中，也必须培养这种抽象能力，这样你的编码水平才会不断提高。
- 2.如果你使用了Netty，并且二进制协议是基于长度的，那么考虑使用LengthFieldBasedFrameDecoder吧，通过调整各种参数，一定会满足你的需求。
- 3.LengthFieldBasedFrameDecoder的拆包包括合法参数校验、异常包处理，以及最后调用ByteBuf的retainedSlice来实现无内存拷贝的拆包。

第25章

ChannelPipeline解析

通过第22章的学习，我们已经知道，Netty的Reactor线程就像是一个发动机，驱动着整个框架的运行，而服务端启动和新连接接入正是发动机的导火线，将发动机点燃。

在服务端端口绑定和新连接建立的过程中会建立相应的Channel，而与Channel密切相关的是ChannelPipeline这个概念，ChannelPipeline可以看作一条流水线，原料（字节流）进来，经过加工，形成一个个Java对象，然后基于这些对象进行处理，最后输出二进制字节流。

本章将以新连接接入为入口，分以下几个部分介绍Netty中的ChannelPipeline是如何运转起来的。

- ChannelPipeline的初始化。
- ChannelPipeline添加ChannelHandler。
- ChannelPipeline删除ChannelHandler。
- Inbound事件的传播。
- Outbound事件的传播。
- 异常事件的传播。

25.1 ChannelPipeline的初始化

在第22章新连接建立的过程中，我们已经知道创建NioSocketChannel的时候会将Netty的核心组件创建出来，ChannelPipeline就是其中的一员。

AbstractChannel.java

```
protected AbstractChannel(Channel parent) {  
    this.parent = parent;  
  
    id = newId();
```

```
unsafe = newUnsafe();

pipeline = newChannelPipeline();

}
```

创建Channel的时候，调用newChannelPipeline()方法创建ChannelPipeline。

AbstractChannel.java

```
protected DefaultChannelPipeline newChannelPipeline() {

    return new DefaultChannelPipeline(this);

}
```

创建一个ChannelPipeline的默认实现如下。

DefaultChannelPipeline.java

```
protected DefaultChannelPipeline(Channel channel) {

    this.channel = ObjectUtil.checkNotNull(channel, "channel");

    tail = new TailContext(this);

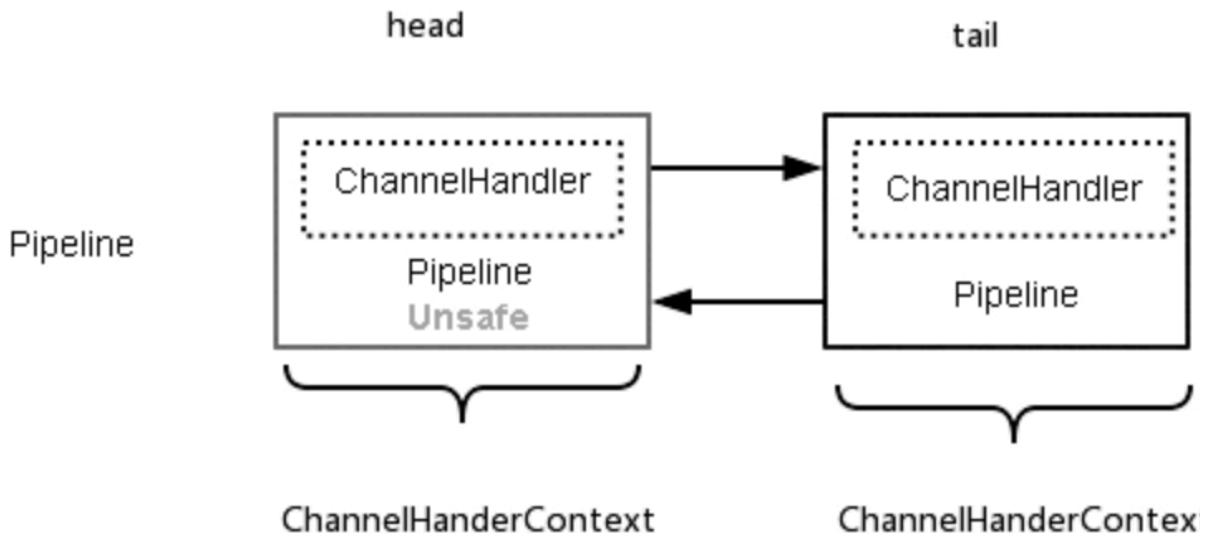
    head = new HeadContext(this);

    head.next = tail;

    tail.prev = head;

}
```

ChannelPipeline中保存了Channel的引用，创建完ChannelPipeline之后，整个ChannelPipeline的结构如下图所示。



`ChannelPipeline`中的每个节点都是一个`ChannelHandlerContext`对象，每个`ChannelHandlerContext`节点都保存了它包裹的执行器`ChannelHandler`执行操作所需要的上下文，其实就是`ChannelPipeline`，因为`ChannelPipeline`包含了`Channel`的引用，所以可以拿到所有的上下文信息。

在默认情况下，一条`ChannelPipeline`会有两个节点：`TailContext`和`HeadContext`。后面我们会具体分析这两个特殊的节点。接下来，我们先分析如何往`ChannelPipeline`中添加一个节点。

25.2 ChannelPipeline添加ChannelHandler

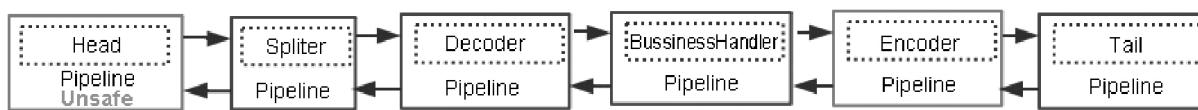
下面是一段非常常见的客户端代码。

```
bootstrap.childHandler(new ChannelInitializer<SocketChannel>() {
    @Override
    public void initChannel(SocketChannel ch) throws Exception {
        ChannelPipeline p = ch.pipeline();
        p.addLast(new Splitter());
        p.addLast(new Decoder());
        p.addLast(new BusinessHandler());
    }
});
```

```
p.addLast(new Encoder());  
}  
});
```

首先，用一个拆包器Splitter将二进制数据流进行拆包，然后将拆出来的包进行解码，解码成Java对象之后传入业务处理器BusinessHandler，业务处理完编码成二进制形式输出。

整个ChannelPipeline结构如下图所示。



这里共有两种不同类型的节点，一种是ChannelInboundHandler，处理Inbound事件，最典型的就是读取数据流，加工处理；还有一种类型的节点是ChannelOutboundHandler，处理Outbound事件，比如当调用writeAndFlush()类方法时，就会经过该种类型的Handler。

不管是哪种类型的Handler，其外层对象ChannelHandlerContext之间都是通过双向链表连接的。而区分一个Handler到底是ChannelInboundHandler还是ChannelOutboundHandler，在添加节点的时候我们就可以看到Netty是怎么处理的。

DefaultChannelPipeline.java

```
@Override  
  
public final ChannelPipeline addLast(ChannelHandler... handlers) {  
  
    return addLast(null, handlers);  
  
}  
  
@Override  
  
public final ChannelPipeline addLast(EventExecutorGroup executor,  
    ChannelHandler...  
}
```

```
handlers) {  
  
    for (ChannelHandler h: handlers) {  
  
        addLast(executor, null, h);  
  
    }  
  
    return this;  
  
}
```

继续调用addLast()方法。

DefaultChannelPipeline.java

```
public final ChannelPipeline addLast(EventExecutorGroup group, String  
name,  
  
ChannelHandler handler) {  
  
    final AbstractChannelHandlerContext newCtx;  
  
    synchronized (this) {  
  
        // 1.检查是否有重复的Handler  
  
        checkMultiplicity(handler);  
  
        // 2.创建节点  
  
        newCtx = newContext(group, filterName(name, handler), handler);  
  
        // 3.添加节点  
  
        addLast0(newCtx);  
  
    }  
  
    // 4.回调用户方法  
  
    callHandlerAdded0(handler);
```

```
    return this;

}
```

这里简单地用synchronized关键词是为了防止多线程并发操作ChannelPipeline底层的双向链表，实际添加节点的过程分为以下4部分。

- 1.检查是否有重复的Handler。
- 2.创建节点。
- 3.添加节点。
- 4.回调用户方法。

接下来，我们分析ChannelPipeline实际添加节点的过程。

25.2.1 检查是否有重复的Handler

DefaultChannelPipeline.java

```
public final ChannelPipeline addLast(EventExecutorGroup group, String
name,
ChannelHandler handler)  {

    final AbstractChannelHandlerContext newCtx;

    synchronized (this)  {

        // 1.检查是否有重复的Handler

        checkMultiplicity(handler);

        // 2.创建节点

        // 3.添加节点

    }

    // 4.回调用户方法
}
```

```
    return this;  
  
}
```

在用户代码中，调用addLast方法添加一个Handler对象的时候，首先会查看该Handler有没有被添加过。

DefaultChannelPipeline.java

```
private static void checkMultiplicity(ChannelHandler handler) {  
  
    if (handler instanceof ChannelHandlerAdapter) {  
  
        ChannelHandlerAdapter h = (ChannelHandlerAdapter) handler;  
  
        if (!h.isSharable() && h.added) {  
  
            throw new ChannelPipelineException(  
                h.getClass().getName() +  
  
                " is not a @Sharable handler, so can't be added or removed multiple  
times.");  
  
        }  
  
        h.added = true;  
  
    }  
  
}
```

Netty使用一个成员变量added标识一个ChannelHandler是否已经添加。上面这段代码很简单，如果当前要添加的Handler是非共享的，并且已经添加过，那么就抛出异常；否则，标识该Handler已经添加。

由此可见，一个Handler如果是支持共享的，就可以无限次被添加到ChannelPipeline中。客户端代码如果要让一个Handler共享，只需要加一个@Sharable注解即可，例如：

```
@Sharable

public class BusinessHandler {  
    }  
}
```

而如果Handler是共享的，一般就通过Spring注入的方式使用，而不需要每次都重新创建。

isSharable方法正是通过该Handler对应的类是否标注@Sharable注解来实现的。

ChannelHandlerAdapter.java

```
public boolean isSharable() {  
  
    Class<?> clazz = getClass();  
  
    Map<Class<?>, Boolean> cache =  
InternalThreadLocalMap.get().handlerSharableCache();  
  
    Boolean sharable = cache.get(clazz);  
  
    if (sharable == null) {  
  
        sharable = clazz.isAnnotationPresent(Sharable.class);  
  
        cache.put(clazz, sharable);  
  
    }  
  
    return sharable;  
}  
}
```

从这里也可以看到，Netty为了性能优化，还使用了ThreadLocal来缓存Handler的状态。在高并发海量连接下，每次有新连接添加Handler都会创建调用此方法，从而优化性能。

25.2.2 创建节点

回到主流程，我们看创建上下文这段代码。

DefaultChannelPipeline.java

```
public final ChannelPipeline addLast(EventExecutorGroup group, String
name,
ChannelHandler handler) {

    final AbstractChannelHandlerContext newCtx;

    synchronized (this) {

        // 1.检查是否有重复的Handler

        // 2.创建节点

        newCtx = newContext(group, filterName(name, handler), handler);

        // 3.添加节点

    }

    // 4.回调用户方法

    return this;

}
```

这里我们需要先分析filterName(name,handler)这个方法，这个方法用于给Handler创建一个唯一性的名字。

DefaultChannelPipeline.java

```
private String filterName(String name, ChannelHandler handler) {

    // 名字为空，就生成一个名字

    if (name == null) {

        return generateName(handler);

    }

}
```

```
// 名字不为空，则直接检查是否有重复的节点

checkDuplicateName(name);

return name;

}
```

显然，在默认情况下，我们传入的name为Null，Netty就生成一个默认的name；否则，检查是否有重名，检查通过则返回。

Netty创建默认name的规则为简单类名#0，下面我们来看下具体是怎么实现的。

DefaultChannelPipeline.java

```
private static final FastThreadLocal<Map<Class<?>, String>> nameCaches
=

new FastThreadLocal<Map<Class<?>, String>>() {
    @Override
    protected Map<Class<?>, String> initialValue() throws Exception {
        return new WeakHashMap<Class<?>, String>();
    }
};

private String generateName(ChannelHandler handler) {
    // 先查看缓存中是否生成过默认name
    Map<Class<?>, String> cache = nameCaches.get();
    Class<?> handlerType = handler.getClass();
    String name = cache.get(handlerType);
    // 没有生成过，就生成一个默认name，加入缓存
}
```

```

if (name == null) {

    name = generateName0(handlerType);

    cache.put(handlerType, name);

}

// 生成后，还要看默认name有没有冲突

if (context0(name) != null) {

    String baseName = name.substring(0, name.length() - 1);

    for (int i = 1; ; i++) {

        String newName = baseName + i;

        if (context0(newName) == null) {

            name = newName;

            break;

        }

    }

}

return name;

}

```

Netty使用一个FastThreadLocal变量来缓存Handler的类和默认名称的映射关系，在生成name的时候，首先查看缓存中有没有生成过默认name（简单类名#0），如果没有生成，就调用generateName0()生成默认name，然后加入缓存。

接下来需要检查name是否和已有的name有冲突，调用context0()方法，查找ChannelPipeline里有没有对应的ChannelHandlerContext节点。

DefaultChannelPipeline.java

```
private AbstractChannelHandlerContext context0(String name) {  
  
    AbstractChannelHandlerContext context = head.next;  
  
    while (context != tail) {  
  
        if (context.name().equals(name)) {  
  
            return context;  
  
        }  
  
        context = context.next;  
  
    }  
  
    return null;  
  
}
```

context0()方法链表遍历每一个ChannelHandlerContext，只要发现某个context的name与待添加的name相同，就返回该context，最后抛出异常，这是一个线性搜索的过程。

所以，如果context0(name)!=null成立，说明现有的ChannelHandlerContext节点链表中已经有了一个默认name，那么就从简单类名#1往上一直找，直到找到一个唯一的name，比如简单类名#3，这是一个不断试探的过程。

当然，如果用户代码在添加Handler的时候指定了一个name，那么要做的仅仅是检查一下是否有重复的节点。

DefaultChannelPipeline.java

```
private void checkDuplicateName(String name) {  
  
    if (context0(name) != null) {
```

```
        throw new IllegalArgumentException("Duplicate handler name: " +  
name);  
  
    }  
  
}
```

处理完name之后，就进入调用newContext方法真正创建context的过程。

DefaultChannelPipeline

```
private AbstractChannelHandlerContext newContext(EventExecutorGroup  
group, String name,  
ChannelHandler handler) {  
  
    return new DefaultChannelHandlerContext(this, childExecutor(group),  
name, handler);  
  
}  
  
private EventExecutor childExecutor(EventExecutorGroup group) {  
  
    if (group == null) {  
  
        return null;  
  
    }  
  
//...  
  
}
```

由前面的调用链得知，group为Null，因此childExecutor(group)也返回Null。

接下来调用DefaultChannelHandlerContext构造方法。

DefaultChannelHandlerContext

```
DefaultChannelHandlerContext(  
    ChannelHandlerContext ctx,  
    EventExecutor executor,  
    ChannelPromise promise)
```

```
    DefaultChannelPipeline pipeline, EventExecutor executor, String
    name,
    ChannelHandler handler)  {

    super(pipeline, executor, name, isInbound(handler),
    isOutbound(handler));

    if (handler == null)  {

        throw new NullPointerException("handler");

    }

    this.handler = handler;

}

在DefaultChannelHandlerContext构造方法中，DefaultChannelHandlerContext将参数
回传到父类，保存Handler的引用，我们进入其父类。
```

AbstractChannelHandlerContext.java

```
AbstractChannelHandlerContext(DefaultChannelPipeline pipeline,
EventExecutor executor,

String name,

boolean inbound, boolean outbound)  {

    this.name = ObjectUtil.checkNotNull(name, "name");

    this.pipeline = pipeline;

    this.executor = executor;

    this.inbound = inbound;

    this.outbound = outbound;

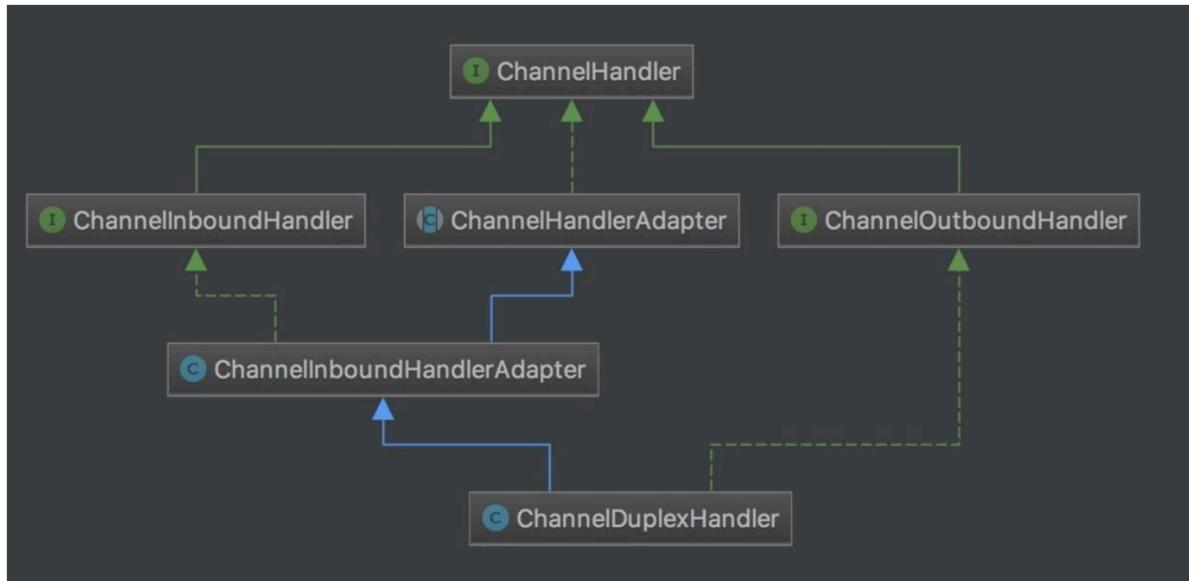
}
```

Netty中用两个字段来表示这个ChannelHandlerContext是Inbound类型还是Outbound类型的，或者两者都是，通过下面两个方法来判断。

DefaultChannelHandlerContext.java

```
private static boolean isInbound(ChannelHandler handler) {  
  
    return handler instanceof ChannelInboundHandler;  
  
}  
  
private static boolean isOutbound(ChannelHandler handler) {  
  
    return handler instanceof ChannelOutboundHandler;  
  
}
```

我们看到，Netty是通过instanceof关键字根据接口类型来判断的，因此，如果一个Handler实现了两类接口，那么它既是一个Inbound类型的Handler，又是一个Outbound类型的Handler，比如ChannelDuplexHandler这个类，如下图所示。



我们常用的将decode操作和encode操作合并到一起的codec，一般会继承MessageToMessageCodec，而MessageToMessageCodec就继承自ChannelDuplexHandler。

MessageToMessageCodec.java

```
public abstract class MessageToMessageCodec<INBOUND_IN, OUTBOUND_IN>
extends

ChannelDuplexHandler {

    protected abstract void encode(ChannelHandlerContext ctx,
OUTBOUND_IN msg,

List<Object> out)

    throws Exception;

    protected abstract void decode(ChannelHandlerContext ctx, INBOUND_IN
msg, List<Object>

out)

    throws Exception;

}
```

ChannelHandlerContext创建完后，接下来需要将这个节点添加到Channel的ChannelPipeline中。

25.2.3 添加节点

DefaultChannelPipeline.java

```
public final ChannelPipeline addLast(EventExecutorGroup group, String
name,

ChannelHandler handler) {

    final AbstractChannelHandlerContext newCtx;

    synchronized (this) {

        // 1.检查是否有重复的Handler

        // 2.创建节点

        // 3.添加节点
```

```
    addLast0(newCtx);

}

// 4.回调用户方法

return this;

}
```

调用addLast0()方法，真正添加节点。

DefaultChannelPipeline.java

```
private void addLast0(AbstractChannelHandlerContext newCtx) {

    AbstractChannelHandlerContext prev = tail.prev;

    newCtx.prev = prev; // 1

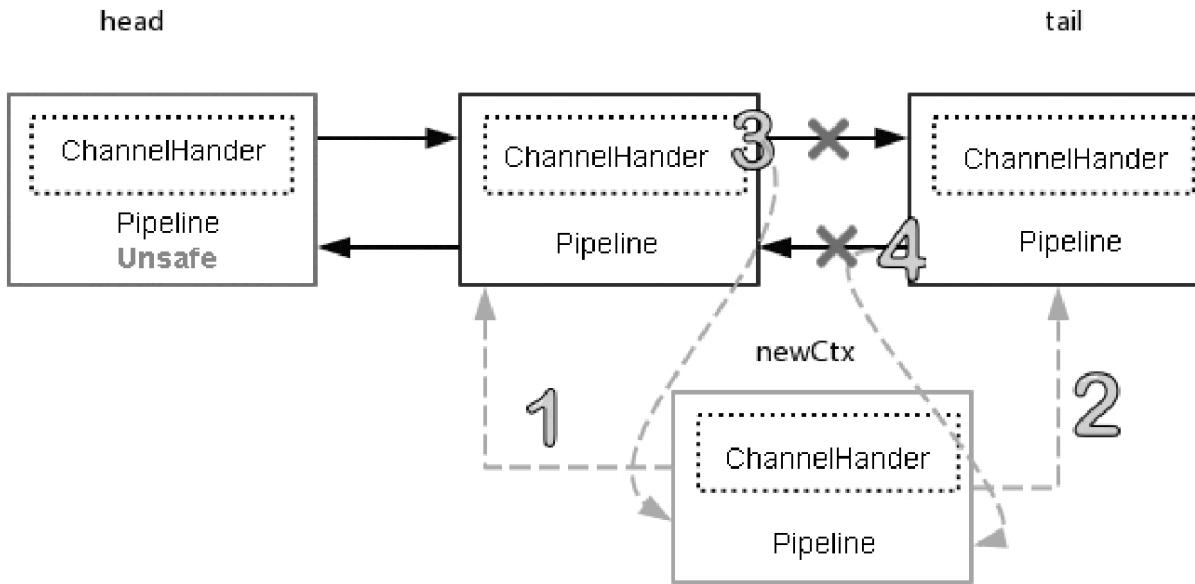
    newCtx.next = tail; // 2

    prev.next = newCtx; // 3

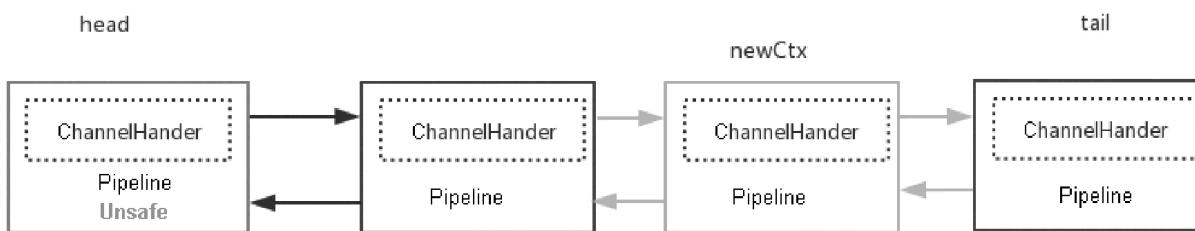
    tail.prev = newCtx; // 4

}
```

用下面这幅图可以简单地表示这个过程，从本质上来说，其实就是一个双向链表的插入操作。



操作完毕，该ChannelHandlerContext节点就加入ChannelHandlerPipeline了，如下图所示。



到这里，ChannelPipeline添加节点的操作就完成了，读者可以根据此思路掌握所有的addXXX()系列方法。

25.2.4 回调用户方法

Netty框架最优秀的设计之一就是在很多地方会埋一些扩展点，用户代码可以在适当的时机做很多定制化操作。这里，当节点添加完毕之后，可以回调用户方法。

DefaultChannelPipeline.java

```
public final ChannelPipeline addLast(EventExecutorGroup group, String
name, ChannelHandler
handler) {
```

```
final AbstractChannelHandlerContext newCtx;

synchronized (this) {

    // 1.检查是否有重复的Handler

    // 2.创建节点

    // 3.添加节点

}

// 4.回调用户方法

callHandlerAdded0(handler);

return this;

}
```

调用callHandlerAdded0。

AbstractChannelHandlerContext.java

```
private void callHandlerAdded0(final AbstractChannelHandlerContext
ctx) {

    ctx.handler().handlerAdded(ctx);

    ctx.setAddComplete();

}
```

到了第4步，ChannelPipeline中的新节点添加完成，于是便开始回调用户代码
ctx.handler().handlerAdded(ctx)，常见的用户代码如下。

AbstractChannelHandlerContext.java

```
public class DemoHandler extends SimpleChannelInboundHandler<...> {

    @Override
```

```
public void handlerAdded(ChannelHandlerContext ctx) throws Exception
{
    // 节点被添加完毕之后回调到此

    // ...

    }
}

}
```

接下来，设置该节点的状态。

AbstractChannelHandlerContext.java

```
final void setAddComplete() {
    for (;;) {
        int oldState = handlerState;
        if (oldState == REMOVE_COMPLETE ||
            HANDLER_STATE_UPDATER.compareAndSet(this,
                oldState, ADD_COMPLETE)) {
            return;
        }
    }
}
```

用CAS修改节点的状态至REMOVE_COMPLETE（说明该节点已经被移除），或者ADD_COMPLETE（添加完成）。

25.2.5 ChannelPipeline添加ChannelHandler小结

用户调用addLast类方法或者其他add类方法往ChannelPipeline中添加ChannelHandler的时候：

1.根据名字检查待添加的Handler是否重复，如果没有名字则使用简单类名默认生成一个名字。

2.创建一个ChannelHandlerContext包裹着Handler，并且每一个ChannelHandlerContext都拥有一个Channel的所有信息。

3.通过双向链表的方式，将ChannelHandlerContext添加到ChannelPipeline中。

4.添加完节点之后，会调用用户所添加Handler的handlerAdded方法。

25.3 ChannelPipeline删除ChannelHandler

Netty最大的特性之一就是ChannelHandler可插拔，做到动态编织ChannelPipeline。

一个比较典型的例子是：在客户端首次连接服务端的时候，需要进行权限认证。认证通过之后，该连接合法，后续就可以不用再认证。

我们只需要使用一个AuthHandler就能满足这个需求：下面是权限认证AuthHandler最简单的实现，第一个数据包传来的是认证信息，如果校验通过，则删除此AuthHandler，后续不会进入verify逻辑；否则，直接关闭连接。

```
public class AuthHandler extends SimpleChannelInboundHandler<ByteBuf>
{
    //...

    @Override
    protected void channelRead0(ChannelHandlerContext ctx, ByteBuf data)
        throws Exception {
        if (verify(authDataPacket)) {
            ctx.pipeline().remove(this);
        } else {
    }
```

```
    ctx.close();

}

}

private boolean verify(ByteBuf byteBuf) {
//...
}

}

}

重点就在ctx.pipeline().remove(this)这段代码。
```

DefaultChannelPipeline.java

```
@Override

public final ChannelPipeline remove(ChannelHandler handler) {
    remove(getContextOrDie(handler));
    return this;
}
```

删除ChannelHandler操作相比添加ChannelHandler简单不少，分为3个步骤。

1.找到待删除的节点。

2.调整双向链表指针并删除。

3.回调用户方法。

25.3.1 找到待删除的节点

DefaultChannelPipeline.java

```
private AbstractChannelHandlerContext getContextOrDie(ChannelHandler
handler) {
```

```
AbstractChannelHandlerContext ctx = (AbstractChannelHandlerContext)
context(handler);

if (ctx == null) {

throw new NoSuchElementException(handler.getClass().getName());

} else {

return ctx;

}

}

}

@Override

public final ChannelHandlerContext context(ChannelHandler handler) {

if (handler == null) {

throw new NullPointerException("handler");

}

AbstractChannelHandlerContext ctx = head.next;

// 遍历双向链表

for (; ; ) {

if (ctx == null) {

return null;

}

if (ctx.handler() == handler) {

return ctx;

}

}
```

```
    ctx = ctx.next;

    }

}
```

这里为了找到Handler对应的ChannelHandlerContext，依然是通过依次遍历双向链表的方式，直到某一个ChannelHandlerContext包裹的Handler和当前Handler相同，即找到了该节点。

25.3.2 调整双向链表指针并删除

找到了ChannelPipeline中对应的ChannelHandlerContext节点之后，即可以展开对这个节点的删除操作了。

DefaultChannelPipeline.java

```
private AbstractChannelHandlerContext remove(final
AbstractChannelHandlerContext ctx)

{

    assert ctx != head && ctx != tail;

    synchronized (this) {

        // 2.调整双向链表指针并删除

        remove0(ctx);

    }

    // 3.回调用户函数

    callHandlerRemoved0(ctx);

    return ctx;

}

private static void remove0(AbstractChannelHandlerContext ctx) {
```

```

AbstractChannelHandlerContext prev = ctx.prev;

AbstractChannelHandlerContext next = ctx.next;

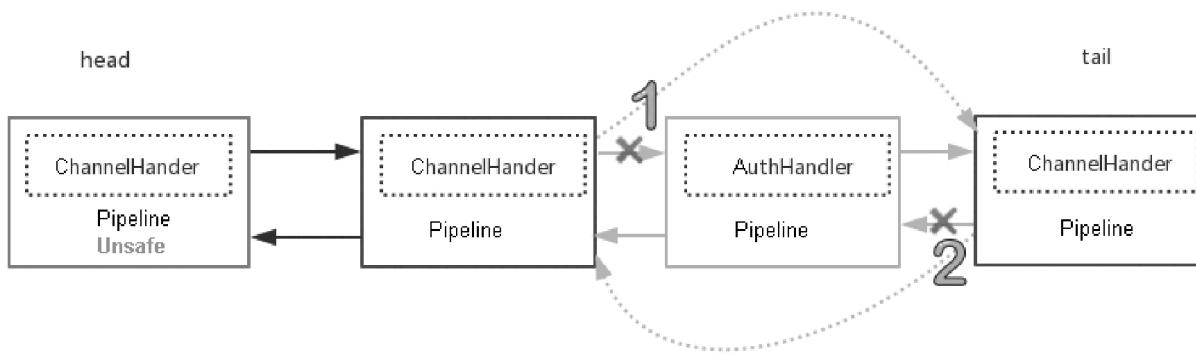
prev.next = next; // 1

next.prev = prev; // 2

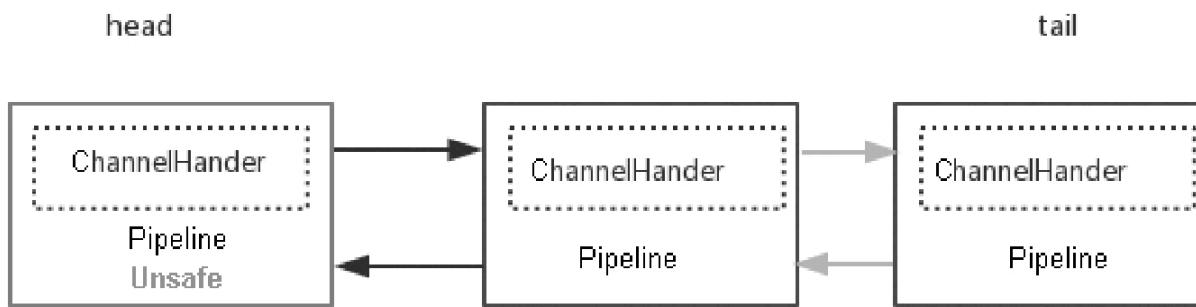
}

```

删除的过程要比添加节点简单，可以用下图来表示。



最后的结果如下图所示。



结合上面两幅图，可以很清晰地了解权限验证AuthHandler的工作原理。另外，被删除的节点因为没有对象引用，过段时间就会被JVM自动回收，删除这个Handler之后，与添加Handler的操作一样，也会有一个用户代码的回调。

25.3.3 回调用户方法

DefaultChannelPipeline.java

```
private void callHandlerRemoved0(final AbstractChannelHandlerContext
ctx) {

    try {

        ctx.handler().handlerRemoved(ctx);

    } finally {

        ctx.setRemoved();

    }

}
```

到了第3步，ChannelPipeline中的节点删除完成，于是便开始回调用户代码
ctx.handler().handlerRemoved(ctx)，常见的用户代码如下。

```
public class DemoHandler extends SimpleChannelInboundHandler<...> {

    @Override

    public void handlerRemoved(ChannelHandlerContext ctx) throws
Exception {

    // 节点被删除完毕后回调到此，可做一些资源清理工作

}

}
```

最后，将该节点的状态设置为REMOVE_COMPLETE。

AbstractChannelHandlerContext.java

```
final void setRemoved() {

    handlerState = REMOVE_COMPLETE;

}
```

removeXX系列的其他方法大同小异，读者可以根据上面的思路展开其他系列方法，这里不再赘述。

25.3.4 ChannelPipeline删除ChannelHandler小结

ChannelPipeline对ChannelHandler的删除和添加是一对相反的操作，删除ChannelHandler的时候：

1. 定位到ChannelHandler对应的ChannelHandlerContext节点。
2. 通过调整ChannelPipeline中双向链表的指针删除对应的ChannelHandlerContext节点。
3. 回调到用户的handlerRemoved方法，我们可以在这个回调中做一些资源清理的操作。

25.4 Inbound事件的传播

我们已经了解了ChannelPipeline在Netty中所处的角色，像一条流水线，控制着字节流的读写。接下来，我们在这个基础上继续深挖Pipeline在事件传播、异常传播等方面的原理。

25.4.1 Unsafe是什么

之所以把Unsafe放到ChannelPipeline中讲，是因为Unsafe和ChannelPipeline密切相关。ChannelPipeline中有关IO的操作最终都是落地到Unsafe的，所以，有必要先讲讲Unsafe。

25.4.1.1 初识Unsafe

顾名思义，Unsafe是不安全的意思，就是告诉你不要在应用程序里直接使用Unsafe及它的衍生类对象。

Netty官方的解释如下：

Unsafe operations that should never be called from user-code.These methods are only provided to implement the actual transport, and must be invoked from an I/O thread.

Unsafe在Channel定义，属于Channel的内部类，表明Unsafe和Channel密切相关。

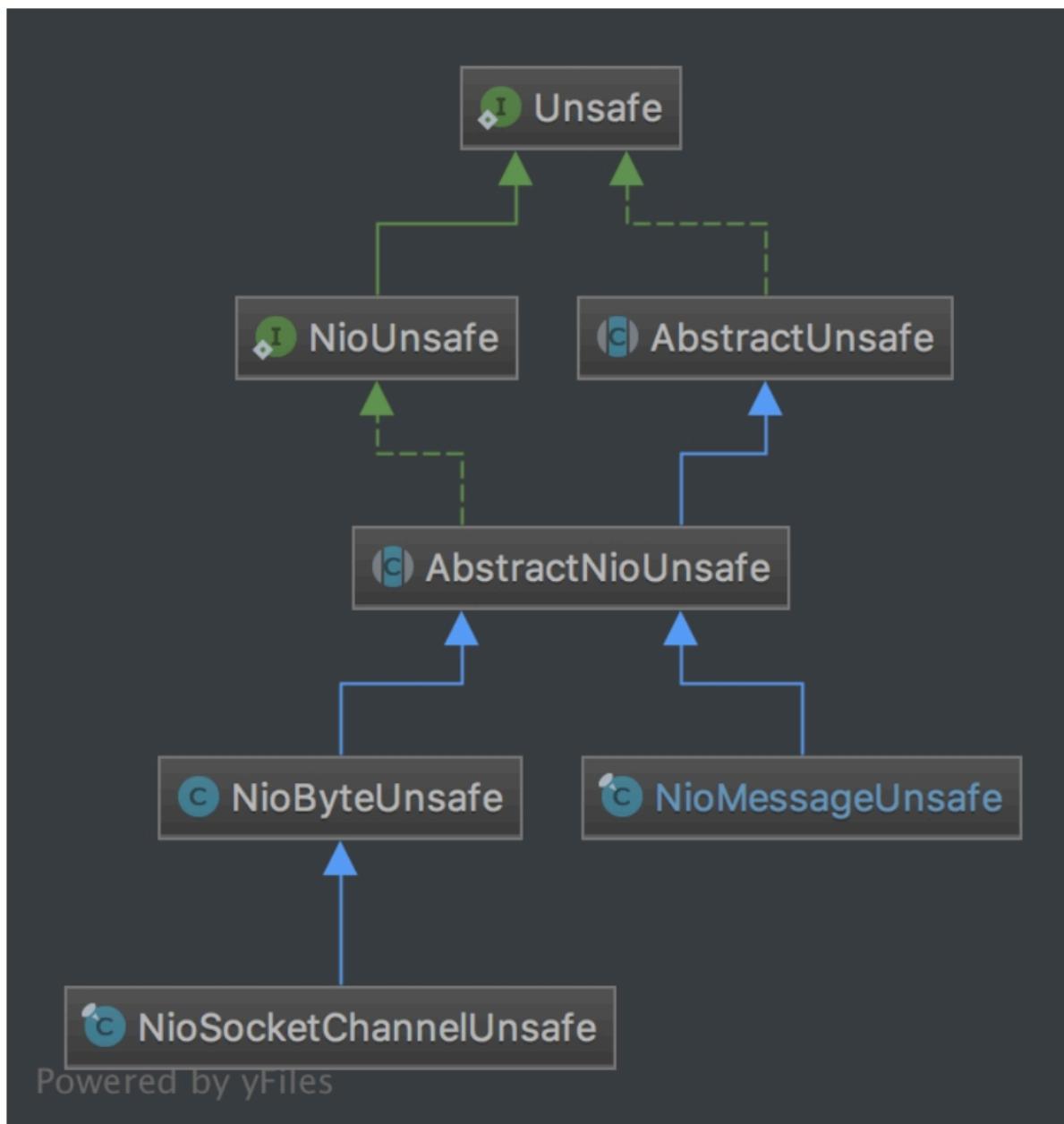
下面是Unsafe接口的所有方法。

```
interface Unsafe {  
  
    RecvByteBufAllocator.Handle recvBufAllocHandle();  
  
    SocketAddress localAddress();  
  
    SocketAddress remoteAddress();  
  
    void register(EventLoop eventLoop, ChannelPromise promise);  
  
    void bind(SocketAddress localAddress, ChannelPromise promise);  
  
    void connect(SocketAddress remoteAddress, SocketAddress  
localAddress, ChannelPromise  
  
promise);  
  
    void disconnect(ChannelPromise promise);  
  
    void close(ChannelPromise promise);  
  
    void closeForcibly();  
  
    void beginRead();  
  
    void write(Object msg, ChannelPromise promise);  
  
    void flush();  
  
    ChannelPromise voidPromise();  
  
    ChannelOutboundBuffer outboundBuffer();  
  
}
```

按功能可以分为分配内存、Socket四元组信息、注册事件循环、绑定端口、Socket的连接和关闭、Socket的读写，看得出来，这些操作都和JDK底层相关。

25.4.1.2 Unsafe继承结构

如下图所示， NioUnsafe在Unsafe基础上增加了以下几个接口。



```
public interface NioUnsafe extends Unsafe {
    SelectableChannel ch();

    void finishConnect();

    void read();
}
```

```
    void forceFlush();  
  
}
```

从增加的接口及类名上来看：

- 1.NioUnsafe增加了可以访问底层JDK的SelectableChannel的功能，定义了从SelectableChannel读取数据的read方法。
- 2.AbstractUnsafe实现了大部分Unsafe的功能。
- 3.AbstractNioUnsafe主要是通过代理到其外部类AbstractNioChannel获得了与JDK NIO相关的一些信息，比如SelectableChannel、SelectionKey等。
- 4.把NioSocketChannelUnsafe和NioByteUnsafe放到一起讲，实现了IO的基本操作——读和写，这些操作都与JDK底层相关。
- 5.NioMessageUnsafe和NioByteUnsafe是处在同一层次的抽象，Netty将一个新连接的建立也当作一个IO操作来处理，这里Message的含义我们可以当作一个SelectableChannel，读的意思就是接收一个SelectableChannel。

25.4.1.3 Unsafe的分类

从以上继承结构来看，我们可以总结出两种类型的Unsafe，一个是与连接的字节数据读写相关的NioByteUnsafe，一个是与新连接建立操作相关的NioMessageUnsafe。

- 1.NioByteUnsafe的读。

AbstractNioByteChannel.java

```
protected class NioByteUnsafe extends AbstractNioUnsafe {  
  
    public final void read() {  
  
        // ...  
  
        doReadBytes(byteBuf);
```

```
// ...
}
}
```

NioByteUnsafe中的读被委托到外部类AbstractNioByteChannel。

AbstractNioByteChannel.java

```
protected int doReadBytes(ByteBuf byteBuf) throws Exception {
    final RecvByteBufAllocator.Handle allocHandle =
unsafe().recvBufAllocHandle();
    allocHandle.attemptedBytesRead(byteBuf.writableBytes());
    return byteBuf.writeBytes(javaChannel(),
allocHandle.attemptedBytesRead());
}
```

可以看到，最后一行已经与JDK底层及Netty中的ByteBuf相关，将JDK的SelectableChannel的字节数据读取到Netty的ByteBuf中。

2.NioMessageUnsafe的读。

AbstractNioMessageChannel.java

```
private final class NioMessageUnsafe extends AbstractNioUnsafe {
    public void read() {
        // ...
        doReadMessages(readBuf);
    }
}
```

```
// ...  
}  
}  
}
```

NioMessageUnsafe中的读最后是委托到外部类NioServerSocketChannel。

NioServerSocketChannel.java

```
protected int doReadMessages(List<Object> buf) throws Exception {  
  
    SocketChannel ch = javaChannel().accept();  
  
    if (ch != null) {  
  
        buf.add(new NioSocketChannel(this, ch));  
  
        return 1;  
  
    }  
  
    return 0;  
}  
}
```

NioMessageUnsafe的读操作很简单，就是调用JDK的accept()方法，新建立一条连接。

3.NioByteUnsafe的写。NioByteUnsafe中的写有两个方法，一个是write，一个是flush。write是将数据添加到Netty的缓冲区，实际将字节流写到TCP缓冲区中的方法是flush，最终会委托到NioSocketChannel的doWrite方法。

NioSocketChannel.java

```
protected void doWrite(ChannelOutboundBuffer in) throws Exception {  
  
    // ...  
  
    SocketChannel ch = javaChannel();  
  
    // ...
```

```
ByteBuffer nioBuffer = nioBuffers [0] ;  
  
// ...  
  
ch.write(nioBuffer);  
  
}
```

这个方法的代码有点多，这里我们只需要关心重点代码即可。我们发现，这个方法最终会调用JDK底层的Channel进行数据读写。

4.NioMessageUnsafe的写。NioMessageUnsafe的写没有太大意义，这里就不分析了。

25.4.2 ChannelPipeline中的HeadContext

HeadContext节点在ChannelPipeline中第一个处理IO事件，新连接接入和读事件在Reactor线程的第二个过程（检测IO事件）中被检测到，我们在第22.3.3节中已经介绍过。

NioEventLoop.java

```
private void processSelectedKey(SelectionKey k, AbstractNioChannel ch)  
{  
  
    final AbstractNioChannel.NioUnsafe unsafe = ch.unsafe();  
  
    //新连接已准备接入或者已存在的连接有数据可读  
  
    if ((readyOps & (SelectionKey.OP_READ | SelectionKey.OP_ACCEPT)) !=  
0 || readyOps  
== 0) {  
  
        unsafe.read();  
  
    }  
  
}
```

读操作直接依赖Unsafe来进行，新连接的接入在前面的章节中已详细阐述过，这里不再描述。下面我们将重点放到连接字节数据流的读写。连接数据读写对应的Unsafe是NioByteUnsafe。

NioByteUnsafe.java

```
@Override

public final void read() {

    final ChannelConfig config = config();

    final ChannelPipeline pipeline = pipeline();

    // 创建ByteBuf分配器

    final ByteBufAllocator allocator = config.getAllocator();

    final RecvByteBufAllocator.Handle allocHandle =
recvBufAllocHandle();

    allocHandle.reset(config);

    ByteBuf byteBuf = null;

    do  {

        // 1. 分配一个 ByteBuf

        byteBuf = allocHandle.allocate(allocator);

        // 2. 将数据读取到分配的 ByteBuf 中

        allocHandle.lastBytesRead(doReadBytes(byteBuf));

        if (allocHandle.lastBytesRead() <= 0)  {

            byteBuf.release();

            byteBuf = null;

            close = allocHandle.lastBytesRead() < 0;
```

```
        break;

    }

// 3. 触发事件，将会引发 ChannelPipeline 的读事件传播

pipeline.fireChannelRead(byteBuf);

byteBuf = null;

} while (allocHandle.continueReading());

pipeline.fireChannelReadComplete();

}
```

同样，笔者抽出了核心代码，先剪去细枝末节，NioByteUnsafe要做的事情可以简单地分为以下几个步骤。

- 1.通过Channel的ChannelConfig，获取ByteBuf分配器，用分配器来分配一个ByteBuf，ByteBuf是Netty里的字节数据载体。
- 2.将Channel中的数据读取到ByteBuf。
- 3.数据读完之后，调用pipeline.fireChannelRead(byteBuf)从HeadContext节点开始传播事件至整个ChannelPipeline。

这里，我们的重点其实就是pipeline.fireChannelRead(byteBuf)。

DefaultChannelPipeline.java

```
final AbstractChannelHandlerContext head;

//...

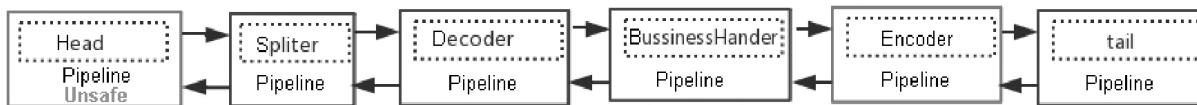
head = new HeadContext(this);

public final ChannelPipeline fireChannelRead(Object msg)  {

    AbstractChannelHandlerContext.invokeChannelRead(head, msg);
```

```
    return this;  
  
}
```

结合下面这幅Pipeline的结构图，可以看到，数据从HeadContext节点开始流入，然后传播至每一个ChannelHandler。在进行下一步分析之前，我们先把HeadContext节点的功能捋一遍。



HeadContext

```
final class HeadContext extends AbstractChannelHandlerContext  
  
implements ChannelOutboundHandler, ChannelInboundHandler {  
  
private final Unsafe unsafe;  
  
HeadContext(DefaultChannelPipeline pipeline) {  
  
super(pipeline, null, HEAD_NAME, false, true);  
  
unsafe = pipeline.channel().unsafe();  
  
setAddComplete();  
  
}  
  
@Override  
  
public ChannelHandler handler() {  
  
return this;  
  
}  
  
@Override
```

```
public void handlerAdded(ChannelHandlerContext ctx) throws Exception
{
    // NOOP
}

@Override

public void handlerRemoved(ChannelHandlerContext ctx) throws
Exception {
    // NOOP
}

@Override

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

@Override

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

```
    @Override

    public void disconnect(ChannelHandlerContext ctx, ChannelPromise
promise) throws
Exception  {

    unsafe.disconnect(promise);

}

@Override

public void close(ChannelHandlerContext ctx, ChannelPromise promise)
throws Exception

{

unsafe.close(promise);

}

@Override

public void deregister(ChannelHandlerContext ctx, ChannelPromise
promise) throws
Exception  {

unsafe.deregister(promise);

}

@Override

public void read(ChannelHandlerContext ctx)  {

unsafe.beginRead();

}
```

```
    @Override

    public void write(ChannelHandlerContext ctx, Object msg,
ChannelPromise promise)

throws Exception {

    unsafe.write(msg, promise);

}

@Override

public void flush(ChannelHandlerContext ctx) throws Exception {

unsafe.flush();

}

@Override

public void exceptionCaught(ChannelHandlerContext ctx, Throwable
cause) throws

Exception {

ctx.fireExceptionCaught(cause);

}

@Override

public void channelRegistered(ChannelHandlerContext ctx) throws
Exception {

invokeHandlerAddedIfNeeded();

ctx.fireChannelRegistered();

}

@Override
```

```
public void channelUnregistered(ChannelHandlerContext ctx) throws
Exception {
    ctx.fireChannelUnregistered();

    if (! channel.isOpen()) {
        destroy();
    }
}

@Override
public void channelActive(ChannelHandlerContext ctx) throws
Exception {
    ctx.fireChannelActive();

    readIfIsAutoRead();
}

@Override
public void channelInactive(ChannelHandlerContext ctx) throws
Exception {
    ctx.fireChannelInactive();
}

@Override
public void channelRead(ChannelHandlerContext ctx, Object msg)
throws Exception {
    ctx.fireChannelRead(msg);
}
```

```
    @Override

    public void channelReadComplete(ChannelHandlerContext ctx) throws
Exception  {

    ctx.fireChannelReadComplete();

    readIfIsAutoRead();

}

private void readIfIsAutoRead()  {

if (channel.config().isAutoRead())  {

    channel.read();

}

}

@Override

public void userEventTriggered(ChannelHandlerContext ctx, Object
evt) throws Exception  {

    ctx.fireUserEventTriggered(evt);

}

@Override

public void channelWritabilityChanged(ChannelHandlerContext ctx)
throws Exception

{
```

```
    ctx.fireChannelWritabilityChanged();

}

}
```

从HeadContext节点继承的两个接口看，它既是一个ChannelHandlerContext（表明它本身就是ChannelPipeline中的一个节点），同时属于Inbound和Outbound类型的Handler。

在传播读写事件的时候，HeadContext的功能只是简单地将事件传播下去，如ctx.fireChannelRead(msg)，在真正执行读写操作的时候，例如在调用writeAndFlush()等方法的时候，最终都会委托Unsafe执行。

当一次数据读完，channelReadComplete方法首先被调用，它要做的事情除了将事件继续传播下去，还会继续向Reactor线程注册读事件，即调用readIfIsAutoRead()，我们简单分析一下。

HeadContext

```
private void readIfIsAutoRead() {
    if (channel.config().isAutoRead()) {
        channel.read();
    }
}
```

调用Channel的读方法。

AbstractChannel.java

```
@Override
public Channel read() {
    pipeline.read();
}
```

```
    return this;

}
```

在默认情况下，Channel都是默认开启自动读取模式的，即只要Channel是活跃的，读完一波数据之后就继续向Selector注册读事件，这样就可以连续不断地读取数据，最终通过ChannelPipeline，传递到HeadContext节点。

HeadContext

```
@Override

public void read(ChannelHandlerContext ctx) {

    unsafe.beginRead();

}
```

接下来委托到了NioByteUnsafe。

NioByteUnsafe.java

```
@Override

public final void beginRead() {

    doBeginRead();

}
```

AbstractNioChannel.java

```
@Override

protected void doBeginRead() throws Exception {

    // Channel.read() or ChannelHandlerContext.read() was called

    final SelectionKey selectionKey = this.selectionKey;

    if (! selectionKey.isValid()) {
```

```

    return;

}

readPending = true;

final int interestOps = selectionKey.interestOps();

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

}
}

```

doBeginRead()做的事情很简单，拿到处理过的SelectionKey，如果发现该SelectionKey在某个地方被移除了readInterestOp操作，这里会给它加上（事实上，通常情况下是不会走到这一行的，即if条件不会成立），只有在TCP三次握手成功之后，才会调用如下方法，下面是TCP三次握手成功之后的回调。

AbstractNioChannel.java

```

public void channelActive(ChannelHandlerContext ctx) throws Exception
{
    ctx.fireChannelActive();

    readIfIsAutoRead();

}

```

在首次连接的时候，会将readInterestOp注册到SelectionKey。

总结一点，HeadContext节点的作用就是作为ChannelPipeline的头节点，开始传递读写事件，调用Unsafe进行实际的读写操作。下面我们开始分析ChannelPipeline中Inbound事件的传播。

25.4.3 ChannelPipeline中的Inbound事件传播

这一节，我们通过ChannelActive事件来分析一下ChannelPipeline中Inbound事件的传播，其他Inbound事件，包括channelInactive、channelRegistered、channelUnregistered、channelRead、channelReadComplete等，原理是一致的。

在第24章中，我们没有详细描述为什么pipeline.fireChannelActive()最终会调用AbstractNio-Channel.doBeginRead()，了解ChannelPipeline中的事件传播机制，你会发现其实相当简单。

新连接建立成功，三次握手成功之后，pipeline.fireChannelActive()被调用。

DefaultChannelPipeline.java

```
public final ChannelPipeline fireChannelActive() {  
  
    AbstractChannelHandlerContext.invokeChannelActive(head);  
  
    return this;  
  
}
```

然后以HeadContext节点为参数，直接调用一个静态方法。

AbstractChannelHandlerContext.java

```
static void invokeChannelActive(final AbstractChannelHandlerContext  
next) {  
  
    EventExecutor executor = next.executor();  
  
    if (executor.inEventLoop()) {  
  
        next.invokeChannelActive();  
  
    } else {  
  
        executor.execute(new Runnable() {  
  
            @Override  
  
            public void run() {  
  
            }  
        });  
    }  
}
```

```
    next.invokeChannelActive();

}

} );

}

}
```

首先，Netty为了确保线程的安全性，将确保该操作在Reactor线程中被执行，因为是在Reactor线程中执行的，所以直接调用HeadContext.fireChannelActive()方法。

HeadContext

```
public void channelActive(ChannelHandlerContext ctx) throws Exception
{
    ctx.fireChannelActive();

    readIfIsAutoRead();

}
```

可以看到，readIfIsAutoRead()在HeadContext传播active事件的时候被调用。

我们继续分析channelActive事件的传播过程。

AbstractChannelHandlerContext.java

```
public ChannelHandlerContext fireChannelActive() {
    final AbstractChannelHandlerContext next = findContextInbound();
    invokeChannelActive(next);
    return this;
}
```

首先，调用`findContextInbound()`找到下一个Inbound节点，由于当前ChannelPipeline的双向链表结构中，既有Inbound节点，又有Outbound节点，让我们看看Netty是如何找到下一个Inbound节点的。

AbstractChannelHandlerContext.java

```
private AbstractChannelHandlerContext findContextInbound() {  
  
    AbstractChannelHandlerContext ctx = this;  
  
    do {  
  
        ctx = ctx.next;  
  
    } while (!ctx.inbound);  
  
    return ctx;  
  
}
```

这段代码很清楚地表明，Netty寻找下一个Inbound节点的过程是一个线性搜索的过程，它会遍历双向链表的下一个节点，直到下一个节点为Inbound。

找到下一个节点之后，递归调用`invokeChannelActive(next);`，直到最后一个Inbound节点——TailContext节点。

TailContext

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

TailContext节点的该方法为空，结束调用。同理，可以分析其他Inbound事件的传播，在正常情况下，即用户如果不覆盖每个节点的事件回调方法，则几乎所有的事件最后都落到TailContext节点。所以，接下来，我们分析一下TailContext节点的功能。

25.4.4 ChannelPipeline中的TailContext

AbstractChannelHandlerContext.java

```
final class TailContext extends AbstractChannelHandlerContext
implements

ChannelInboundHandler {

    TailContext(DefaultChannelPipeline pipeline) {
        super(pipeline, null, TAIL_NAME, true, false);
        setAddComplete();
    }

    @Override
    public ChannelHandler handler() {
        return this;
    }

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

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

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

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

    @Override
```

```
    public void channelWritabilityChanged(ChannelHandlerContext ctx)
throws Exception

    {

@Override

    public void handlerAdded(ChannelHandlerContext ctx) throws Exception
{ }

@Override

    public void handlerRemoved(ChannelHandlerContext ctx) throws
Exception { }

@Override

    public void userEventTriggered(ChannelHandlerContext ctx, Object
evt) throws
Exception {

    ReferenceCountUtil.release(evt);

}

@Override

    public void exceptionCaught(ChannelHandlerContext ctx, Throwable
cause) throws
Exception {

    onUnhandledInboundException(cause);

}

@Override

    public void channelRead(ChannelHandlerContext ctx, Object msg)
throws Exception { }
```

```
    onUnhandledInboundMessage(msg);

}

@Override

public void channelReadComplete(ChannelHandlerContext ctx) throws
Exception { }

}
```

正如前面提到的，TailContext节点的大部分作用为终止事件的传播（方法体为空）。除此之外，有两个重要的方法我们必须提一下，即exceptionCaught和channelRead。

首先来看一下exceptionCaught这个方法。

TailContext

```
@Override

public void exceptionCaught(ChannelHandlerContext ctx, Throwable
cause) throws Exception

{

    onUnhandledInboundException(cause);

}
```

异常传播的机制和Inbound事件传播的机制类似，最终如果用户自定义ChannelHandler没有处理，则会落到TailContext节点，TailContext节点可不会简单地“吞下”这个异常，而是发出警告，相信大家对下面这段警告不陌生吧？

TailContext

```
protected void onUnhandledInboundException(Throwable cause) {

    try {

        logger.warn(

```

```
"An exceptionCaught() event was fired, and it reached at the tail of
the
pipeline. " +
"It usually means the last handler in the pipeline did not handle
the exception.",
cause);
} finally {
ReferenceCountUtil.release(cause);
}
}
```

我们再来看一下channelRead方法。

TailContext

```
@Override
public void channelRead(ChannelHandlerContext ctx, Object msg) throws
Exception {
onUnhandledInboundMessage(msg);
}
```

TailContext节点在发现字节数据（ByteBuf）或者decode之后的业务对象在ChannelPipeline流转过程中没有被消费，落到TailContext节点时，TailContext节点就会发出一个警告，告诉你：“我已经将你未处理的数据丢掉了！”

```
protected void onUnhandledInboundMessage(Object msg) {
try {
logger.debug(

```

```
"Discarded inbound message {} that reached at the tail of the  
pipeline. " +  
  
"Please check your pipeline configuration.", msg);  
  
} finally {  
  
    ReferenceCountUtil.release(msg);  
  
}  
  
}
```

发出警告之后，默认会将未处理的对象尽心释放。

总结一下，TailContext节点的作用就是结束事件传播，并且对一些重要的事件进行善意提醒。

25.4.5 Inbound事件的传播小结

- 1.一般我们的自定义ChannelInboundHandler都继承自ChannelInboundHandlerAdapter类，如果用户代码没有覆盖ChannelInboundHandler.channelXXX方法，Inbound事件从HeadContext开始传播，遍历ChannelPipeline的双向链表，默认情况下传递到TailContext节点。
- 2.如果用户代码覆盖了ChannelInboundHandler.channelXXX方法，那么事件传播就在当前节点结束。
- 3.如果用户代码调用ChannelHandlerContext.fireXXX来传播事件，那么这个事件就从当前节点开始往下传播。

25.5 Outbound事件的传播

在这一节中，我们以最常见的writeAndFlush方法调用来分析ChannelPipeline中的Outbound事件是如何传播的。

在典型的推送系统中，会有类似下面的一段代码。

用户代码

```
Channel channel = ChannelManager.getChannel(userId);
```

```
channel.writeAndFlush(response);
```

这段代码的含义就是根据用户ID获得对应的Channel，然后向用户推送消息，跟进channel.writeAndFlush()。

NioSocketChannel.java

```
public ChannelFuture writeAndFlush(Object msg) {  
  
    return pipeline.writeAndFlush(msg);  
  
}
```

从ChanelPipeline开始传播。

DefaultChannelPipeline.java

```
public final ChannelFuture writeAndFlush(Object msg) {  
  
    return tail.writeAndFlush(msg);  
  
}
```

从上面这段代码我们看到，如果通过Channel来传播事件，是从TailContext开始传播，writeAndFlush()方法是TailContext类继承来的方法。

AbstractChannelHandlerContext.java

```
public ChannelFuture writeAndFlush(Object msg) {  
  
    return writeAndFlush(msg, newPromise());  
  
}  
  
public ChannelFuture writeAndFlush(Object msg, ChannelPromise promise)  
{  
  
    write(msg, true, promise);  
  
    return promise;  
  
}
```

Netty中很多IO操作都是异步操作，返回一个ChannelFuture给调用方，调用方获得这个Future后，可以在适当的时机拿到操作的结果，或者注册回调。

我们继续分析write方法。

AbstractChannelHandlerContext.java

```
private void write(Object msg, boolean flush, ChannelPromise promise)
{
    AbstractChannelHandlerContext next = findContextOutbound();

    final Object m = pipeline.touch(msg, next);

    EventExecutor executor = next.executor();

    if (executor.inEventLoop()) {
        if (flush) {
            next.invokeWriteAndFlush(m, promise);
        } else {
            next.invokeWrite(m, promise);
        }
    } else {
        AbstractWriteTask task;

        if (flush) {
            task = WriteAndFlushTask.newInstance(next, m, promise);
        } else {
            task = WriteTask.newInstance(next, m, promise);
        }
    }
}
```

```
safeExecute(executor, task, promise, m);

    }

}
```

Netty为了保证程序的高效执行，所有的核心操作都在Reactor线程中处理，如果业务线程调用Channel的方法，Netty会将该操作封装成一个Task，随后在Reactor线程事件循环的第三个过程中执行。

write先调用findContextOutbound()方法找到下一个Outbound节点。

AbstractChannelHandlerContext

```
private AbstractChannelHandlerContext findContextOutbound() {
    AbstractChannelHandlerContext ctx = this;
    do {
        ctx = ctx.prev;
    } while (!ctx.outbound);
    return ctx;
}
```

找Outbound节点的过程和找Inbound节点类似，反方向遍历ChannelPipeline中的双向链表，直到第一个Outbound节点。

无论调用writeAndFlush方法的线程是Reactor线程还是用户线程，最后都会调用next.invokeWriteAndFlush(m,promise)。

AbstractChannelHandlerContext.java

```
private void invokeWriteAndFlush(Object msg, ChannelPromise promise) {
    if (invokeHandler()) { // 默认返回 true
        invokeWrite0(msg, promise);
    }
}
```

```
    invokeFlush0();

} else {

    writeAndFlush(msg, promise);

}

}
```

writeAndFlush方法到这里被分解为invokeWrite0和invokeFlush0方法， invokeFlush0 和invokeWrite0的传播过程类似。因此，下面我们只分析invokeWrite0方法。

AbstractChannelHandlerContext.java

```
private void invokeWrite0(Object msg, ChannelPromise promise) {

    try {

        ((ChannelOutboundHandler) handler()).write(this, msg, promise);

    } catch (Throwable t) {

        notifyOutboundHandlerException(t, promise);

    }

}
```

我们在使用Outbound类型的ChannelHandler时，一般会继承
ChannelOutboundHandlerAdapter，而ChannelOutboundHandlerAdapter和
ChannelInboundHandlerAdapter的原理类似，默认情况下都会把事件继续传播下去。

ChannelOutboundHandlerAdapter.java

```
public void write(ChannelHandlerContext ctx, Object msg,
    ChannelPromise promise) throws

Exception {
```

```
    ctx.write(msg, promise);

}
```

我们已经知道，在ChannelPipeline的双向链表结构中，最后一个Outbound节点是HeadContext节点，因此数据最终会落地到它的write方法。

HeadContext

```
public void write(ChannelHandlerContext ctx, Object msg,
ChannelPromise promise) throws

Exception {

    unsafe.write(msg, promise);

}
```

这里，其实可以加深我们对HeadContext节点的理解，即所有的数据写出都会经过HeadContext节点，我们在第26章会深入分析写数据过程。

在实际应用程序中，Outbound类的节点中会有一种特殊类型的节点叫encoder，它的作用是根据自定义编码规则将业务对象转换成ByteBuf，而这类Encoder一般继承自MessageToByteEncoder，下面是一段示例代码。

用户代码

```
public abstract class DataPacketEncoder extends
MessageToByteEncoder<DatePacket> {

    @Override

    protected void encode(ChannelHandlerContext ctx, DatePacket msg,
ByteBuf out)

    throws Exception {
```

```
// 这里获得业务对象msg的数据，然后调用 out.writeXXX()系列方法编码  
}  
}  
}
```

为什么业务代码只需要覆盖这里的encode方法，就可以将业务对象转换成字节流写出去呢？我们查看一下其父类MessageToByteEncoder的write方法是怎么处理业务对象的。

MessageToByteEncoder.java

```
public void write(ChannelHandlerContext ctx, Object msg,  
ChannelPromise promise) throws  
Exception {  
  
    ByteBuf buf = null;  
  
    try {  
  
        // 1. 需要判断当前编码器能否处理这类对象  
  
        if (acceptOutboundMessage(msg)) {  
  
            I cast = (I) msg;  
  
            // 2. 分配内存  
  
            buf = allocateBuffer(ctx, cast, preferDirect);  
  
            try {  
  
                // 3. 填充数据  
  
                encode(ctx, cast, buf);  
  
            } finally {  
  
                ReferenceCountUtil.release(cast);  
  
            }  
        }  
    } catch (Exception e) {  
        promise.setException(e);  
    }  
}
```

```
// 4. buf 到这里已经装载着数据，于是把该buf向前传播，直到 HeadContext 节点

if (buf.isReadable()) {

    ctx.write(buf, promise);

} else {

    buf.release();

    ctx.write(Unpooled.EMPTY_BUFFER, promise);

}

buf = null;

} else {

// 5. 如果不能处理，就将Outbound事件继续向前传播

ctx.write(msg, promise);

}

} catch (EncoderException e) {

throw e;

} catch (Throwable e) {

throw new EncoderException(e);

} finally {

// 6. 释放内存

if (buf != null) {
```

```
buf.release();  
  
    }  
  
}
```

1.调用acceptOutboundMessage方法进行判断，该Encoder是否可以处理msg对应的Java对象。通过之后，就强制转换，这里的泛型I对应的是DataPacket。

2.转换之后，开辟一段内存ByteBuf。

3.调用encode()，即回到DataPacketEncoder中，将buf装满数据。

4.如果buf中被写了数据 (buf.isReadable())，就将该buf往前传递，一直传递到HeadContext节点，被HeadContext节点的Unsafe消费掉。

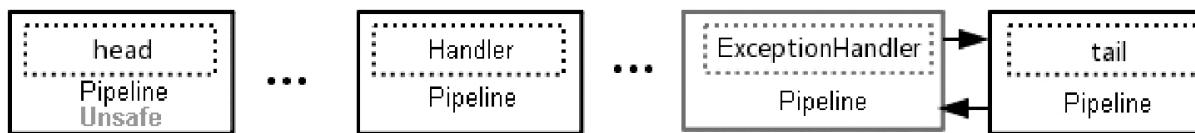
5.当然，如果当前Encoder不能处理当前业务对象，就将该业务对象向前传播，直到HeadContext节点。

6.都处理完之后，释放buf，避免堆外内存泄漏。

最后，我们来简单总结一下：Outbound事件传播机制和Inbound事件传播非常类似，只不过Outbound事件是从链表尾部开始向前传播的，而Inbound事件传播是从链表头部开始传播的。关于写数据，最终都会落到HeadContext节点中的Unsafe来处理。

25.6 ChannelPipeline中异常的传播

我们通常在业务代码中，会加入一个异常处理器，统一处理ChannelPipeline过程中的所有异常，并且，一般该异常处理器需要加在自定义节点的末尾。如下图所示，这个双向链表是我们在应用程序中比较常见的结构。



此类ExceptionHandler一般继承自ChannelDuplexHandler，标识该节点既是一个Inbound节点，又是一个Outbound节点。我们分别分析一下在Inbound事件和Outbound事件过程中，ExceptionHandler是如何处理这些异常的。

25.6.1 Inbound异常的传播

我们以数据的读取为例，看下Netty如何传播在这个过程中发生的异常。

25.6.1.1 Inbound异常的产生

对于每一个节点的数据读取都会调用
AbstractChannelHandlerContext.invokeChannelRead()方法。

AbstractChannelHandlerContext.java

```
private void invokeChannelRead(Object msg) {  
  
    try {  
  
        ((ChannelInboundHandler) handler()).channelRead(this, msg);  
  
    } catch (Throwable t) {  
  
        notifyHandlerException(t);  
  
    }  
  
}
```

可以看到，该ChannelHandlerContext节点最终委托其内部的ChannelHandler来处理channelRead，而在最外层捕获整个Throwable，因此，我们在如下用户代码中的异常会被捕获。

用户代码

```
public class BusinessHandler extends ChannelInboundHandlerAdapter {  
  
    @Override  
  
    protected void channelRead(ChannelHandlerContext ctx, Object data)  
    throws Exception  
  
    {  
  
        //...  
  
        throw new BusinessException(...);  
    }  
}
```

```
//...  
    }  
}  
}
```

上面这段业务代码中的BusinessException会被BusinessHandler所在的节点捕获，进入notifyHandlerException(t)，然后传播。下面我们看下它是如何传播的。

25.6.1.2 Inbound异常的传播

AbstractChannelHandlerContext.java

```
private void notifyHandlerException(Throwable cause) {  
  
    // ...  
  
    invokeExceptionCaught(cause);  
  
}  
  
private void invokeExceptionCaught(final Throwable cause) {  
  
    handler().exceptionCaught(this, cause);  
  
}
```

可以看到，此Handler中的异常优先由此Handler中的exceptionCaught方法来处理，默认情况下，如果不覆写此Handler中的exceptionCaught方法，则会调用ChannelInboundHandlerAdapter的exceptionCaught方法。

ChannelInboundHandlerAdapter.java

```
public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)  
  
throws Exception {  
  
    ctx.fireExceptionCaught(cause);  
  
}
```

AbstractChannelHandlerContext

```
public ChannelHandlerContext fireExceptionCaught(final Throwable cause) {  
  
    invokeExceptionCaught(next, cause);  
  
    return this;  
  
}
```

到了这里，其实已经很清楚了，如果我们在自定义Handler中没有处理异常，那么默认情况下该异常将一直传递下去，遍历每一个节点，直到最后一个自定义异常处理器ExceptionHandler来终结这个异常。

Exceptionhandler

```
public Exceptionhandler extends ChannelDuplexHandler {  
  
    @Override  
  
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)  
  
        throws Exception {  
  
        // 处理该异常，并终止异常的传播  
  
    }  
  
}
```

到了这里，读者应该可以知道为什么异常处理器要加在ChannelPipeline链表的最后了吧？如果加在中间，那么这个异常处理器后面的Handler抛出来的异常都处理不了。

25.6.2 Outbound异常的传播

25.6.2.1 Outbound异常的产生

对于Outbound事件传播过程中所发生的异常，加到ChannelPipeline中双向链表最后的Exceptionhandler也能处理，为什么？

我们以前面提到的writeAndFlush方法为例，来看看Outbound事件传播过程中的异常最后是如何落到ExceptionHandler中去的。

前面我们已经知道，channel.writeAndFlush()方法最终也会调用AbstractChannelHandlerContext的invokeFlush0()方法。

AbstractChannelHandlerContext.java

```
private void invokeWriteAndFlush(Object msg, ChannelPromise promise) {  
  
    if (invokeHandler()) {  
  
        invokeWrite0(msg, promise);  
  
        invokeFlush0();  
  
    } else {  
  
        writeAndFlush(msg, promise);  
  
    }  
  
}  
  
}
```

而invokeFlush0()会委托其内部的ChannelHandler的flush方法。

AbstractChannelHandlerContext.java

```
private void invokeFlush0() {  
  
    try {  
  
        ((ChannelOutboundHandler) handler()).flush(this);  
  
    } catch (Throwable t) {  
  
        notifyHandlerException(t);  
  
    }  
  
}
```

在这个过程中，可能会产生异常。

25.6.2.2 Outbound异常的处理

假设当前节点在flush的过程中发生了异常，会被捕获，进入notifyHandlerException(t)方法来处理，该方法和Inbound事件传播过程中的异常传播方法一样，也是轮流找下一个异常处理器。而如果异常处理器在ChannelPipeline最后面，则一定会被执行到，这就是为什么该异常处理器也能处理Outbound异常的原因。

25.6.2.3 ChannelPipeline中异常的传播小结

关于为什么ExceptionHandler既能处理传播Inbound事件过程中的异常，又能处理传播Outbound事件过程中的异常，总结一点就是：在任何节点中发生的异常都会向下一个节点传递，最后终究会传递到异常处理器。

25.7 总结

最后，我们对本章内容做一个总结。

- 1.以新连接的接入流程为例，在新连接创建的过程中创建了Channel，而在创建Channel的过程中创建了该Channel对应的ChannelPipeline。
- 2.创建完ChannelPipeline之后，给该ChannelPipeline添加了两个节点HeadContext和TailContext，每个节点的数据结构都是ChannelHandlerContext类型的，ChannelHandlerContext中拥有ChannelPipeline和Channel所有的上下文信息。
- 3.ChannelPipeline是双向链表结构，添加和删除节点均只需要调整链表结构。
- 4.ChannelPipeline中的每个节点都包着具体的处理器ChannelHandler，节点根据ChannelHandler的类型是ChannelInboundHandler还是ChannelOutboundHandler来判断该节点属于Inbound类型，还是Outbound类型，或者两者都是。
- 5.一个Channel对应一个Unsafe，Unsafe处理底层IO操作，NioServerSocketChannel对应NioMessageUnsafe，NioSocketChannel对应NioByteUnsafe。
- 6.Inbound事件从HeadContext节点传播到TailContext节点，Outbound事件从TailContext节点传播到HeadContext节点。
- 7.异常在ChannelPipeline中的双向链表中传播时，无论Inbound节点还是Outbound节点，都是向下一个节点传播，直到TailContext节点。

第26章

writeAndFlush解析

在前面的章节中，我们已经详细描述了事件和异常传播在Netty中的实现，其中有一类事件我们在实际编码中用得最多，那就是write或者writeAndFlush。

本章分以下几个部分阐述一个Java对象最后是如何转变成字节流，又写到Socket缓冲区中的。

1.Pipeline中的标准链表结构。

2.Java对象编码过程。

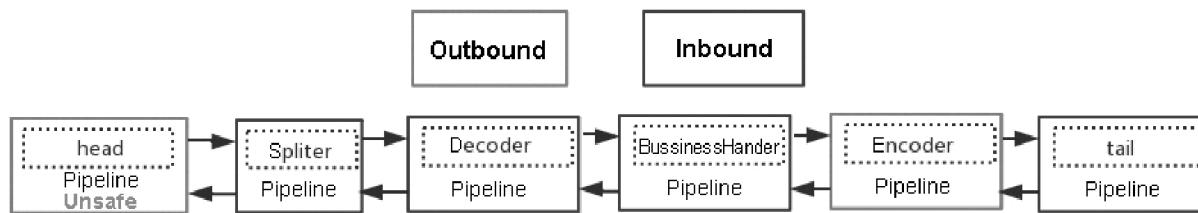
3.write：写队列。

4.flush：刷新写队列。

5.writeAndFlush：写队列并刷新。

26.1 Pipeline中的标准链表结构

一个标准的Pipeline链式结构如下图所示（我们省去了异常处理Handler）。



数据从head节点流入，先拆包，然后解码成业务对象，最后经过业务Handler处理，调用write，将结果对象写出去。而写的过程先通过tail节点，然后通过Encoder节点将对象编码成ByteBuf，最后将该ByteBuf对象传递到head节点，调用底层的Unsafe写到JDK底层管道。

26.2 Java对象编码过程

为什么我们在Pipeline中添加了Encoder节点，Java对象就转换成Netty可以处理的ByteBuf，写到管道里？

我们先看下调用write的代码。

BusinessHandler.java

```
protected void channelRead0(ChannelHandlerContext ctx, Request
request) throws Exception

{

    Response response = doBusiness(request);

    if (response != null) {

        ctx.channel().write(response);

    }

}
```

业务处理器接收请求之后，先做一些业务处理，返回一个Response；然后Response在Pipeline中传递，落到Encoder节点。下面是Encoder节点的处理流程。

Encoder

```
public class Encoder extends MessageToByteEncoder<Response> {

    @Override

    protected void encode(ChannelHandlerContext ctx, Response response,
ByteBuf out)

    throws Exception {

        out.writeByte(response.getVersion());

        out.writeInt(4 + response.getData().length);

    }
}
```

```
    out.writeBytes(response.getData());  
  
    }  
  
}
```

Encoder的处理流程很简单，按照简单的自定义协议，将Java对象Response写到传入的参数out中，这个out到底是什么？

为了回答这个问题，我们需要了解Response对象，从BusinessHandler传入MessageToByteEncoder的时候，首先是传入write方法。

MessageToByteEncoder

```
@Override  
  
public void write(ChannelHandlerContext ctx, Object msg,  
ChannelPromise promise) throws  
Exception {  
  
    ByteBuf buf = null;  
  
    try {  
  
        // 判断当前Handler是否能处理写入的消息  
  
        if (acceptOutboundMessage(msg)) {  
  
            @SuppressWarnings("unchecked")  
  
            // 强制转换  
  
            I cast = (I) msg;  
  
            // 分配一段ByteBuf  
  
            buf = allocateBuffer(ctx, cast, preferDirect);  
  
            try {  
  
                // 调用encode，这里就调回到Encoder这个Handler中  
  
            }
```

```
encode(ctx, cast, buf);

} finally {

// 既然自定义Java对象转换成ByteBuf了，那么这个对象就已经无用了，释放掉

// (当传入的msg类型是ByteBuf的时候，就不需要自己手动释放了)

ReferenceCountUtil.release(cast);

}

// 如果buf中写入了数据，就把buf传到下一个节点

if (buf.isReadable()) {

ctx.write(buf, promise);

} else {

// 否则，释放buf，将空数据传到下一个节点

buf.release();

ctx.write(Unpooled.EMPTY_BUFFER, promise);

}

buf = null;

} else {

// 如果当前节点不能处理传入的对象，直接传递给下一个节点处理

ctx.write(msg, promise);

}

} catch (EncoderException e) {

throw e;
```

```
    } catch (Throwable e) {  
  
        throw new EncoderException(e);  
  
    } finally {  
  
        // 当buf在Pipeline中处理完之后，释放  
  
        if (buf != null) {  
  
            buf.release();  
  
        }  
  
    }  
  
}
```

其实，这一节的内容，在前面的章节中已经提到过，这里我们详细阐述一下Encoder是如何处理传入的Java对象的。

- 1.判断当前Handler是否能处理写入的消息，如果能处理，则进入下面的流程，否则直接传递给下一个节点处理。
- 2.将对象强制转换成Encoder可以处理的Response对象。
- 3.分配一个ByteBuf。
- 4.调用Encoder，即进入Encoder的encode方法，该方法是用户代码，用户将数据写入ByteBuf。
- 5.既然自定义Java对象转换成ByteBuf了，那么这个对象就已经无用了，释放掉（当传入的msg类型是ByteBuf的时候，就不需要自己手动释放了）。
- 6.如果buf中写入了数据，就把buf传到下一个节点，否则释放buf，将空数据传到下一个节点。
- 7.最后，当buf在Pipeline中处理完之后，释放节点。

总结一点就是，Encoder节点分配一个ByteBuf，调用encode方法，将Java对象根据自定义协议写入ByteBuf，然后把ByteBuf传入下一个节点，在我们的例子中，最终会传

入head节点。

HeadContext

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

这里的msg就是前面在Encoder节点中，载有Java对象数据的自定义ByteBuf对象，进入下一节。

26.3 write：写队列

AbstractChannel.java

```
// 下面的方法在AbstractUnsafe类中  
  
@Override  
  
public final void write(Object msg, ChannelPromise promise) {  
  
    assertEventLoop();  
  
    ChannelOutboundBuffer outboundBuffer = this.outboundBuffer;  
  
    int size;  
  
    try {  
  
        msg = filterOutboundMessage(msg);  
  
        size = pipeline.estimatorHandle().size(msg);  
  
        if (size < 0) {
```

```

size = 0;

}

} catch (Throwable t) {

safeSetFailure(promise, t);

ReferenceCountUtil.release(msg);

return;

}

outboundBuffer.addMessage(msg, size, promise);

}

```

1. 调用assertEventLoop确保该方法的调用在Reactor线程中。

2. 调用filterOutboundMessage()方法，将待写入的对象过滤，把非ByteBuf对象和FileRegion过滤，把所有的非直接内存转换成直接内存DirectBuffer。

AbstractNioByteChannel.java

```

@Override

protected final Object filterOutboundMessage(Object msg) {

    if (msg instanceof ByteBuf) {

        ByteBuf buf = (ByteBuf) msg;

        if (buf.isDirect()) {

            return msg;

        }

        return newDirectBuffer(buf);

    }

}

```

```
if (msg instanceof FileRegion) {  
  
    return msg;  
  
}  
  
throw new UnsupportedOperationException(  
  
    "unsupported message type: " + StringUtil.simpleClassName(msg) +  
EXPECTED_TYPES);  
  
}
```

3.估算出需要写入的ByteBuf的size。

4.调用ChannelOutboundBuffer的addMessage(msg,size,promise)方法。

接下来，我们需要重点看一下这个方法干了什么事情。

ChannelOutboundBuffer

```
public void addMessage(Object msg, int size, ChannelPromise promise) {  
  
    // 创建一个待写出的消息节点  
  
    Entry entry = Entry.newInstance(msg, size, total(msg), promise);  
  
    if (tailEntry == null) {  
  
        flushedEntry = null;  
  
        tailEntry = entry;  
  
    } else {  
  
        Entry tail = tailEntry;  
  
        tail.next = entry;  
  
        tailEntry = entry;  
  
    }  
}
```

```

if (unflushedEntry == null) {

    unflushedEntry = entry;

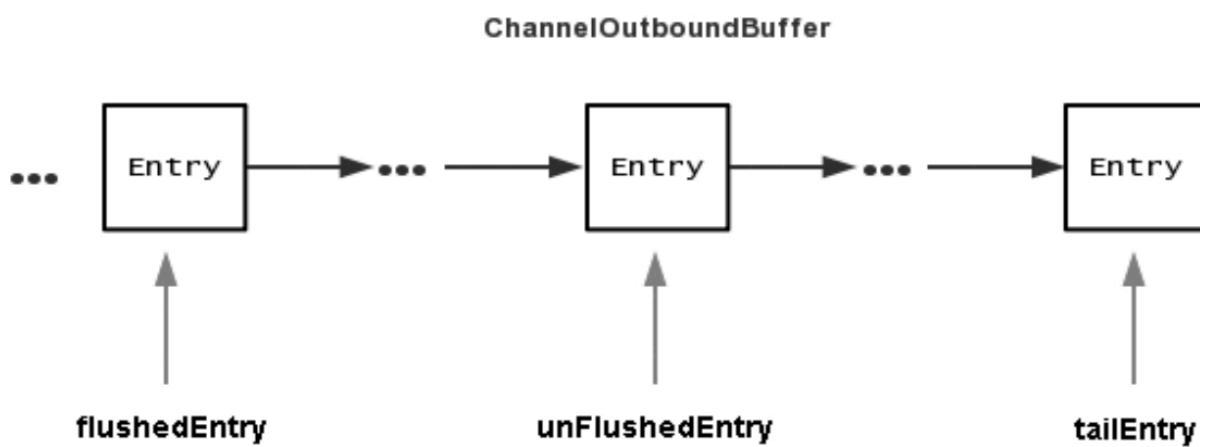
}

incrementPendingOutboundBytes(size, false);

}

```

想要理解上面这段代码，必须得掌握写缓存中的几个消息指针，如下图所示。



ChannelOutboundBuffer里的数据结构是一个单链表结构，每个节点都是一个Entry，Entry里包含了待写出ByteBuf及消息回调promise，下面分别是三个指针的作用。

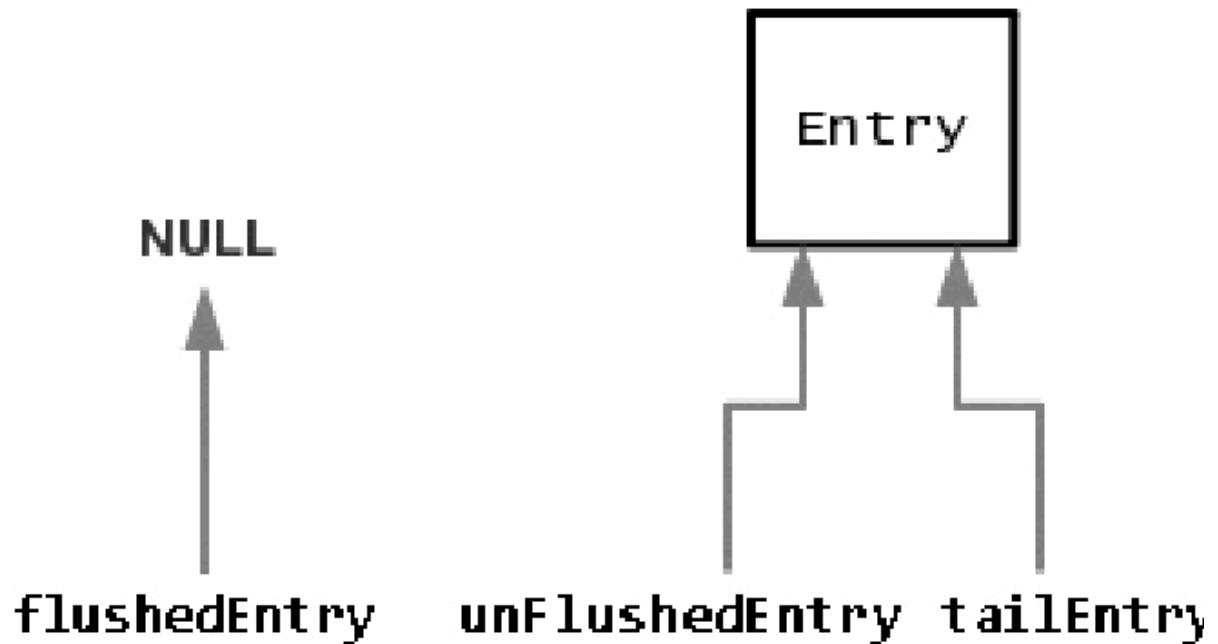
1.**flushedEntry**指针表示第一个被写入操作系统Socket缓冲区的节点。

2.**unFlushedEntry**指针表示第一个未被写入操作系统Socket缓冲区的节点。

3.**tailEntry**指针表示ChannelOutboundBuffer缓冲区的最后一个节点。

初次调用addMessage之后，各个指针的情况如下图所示。

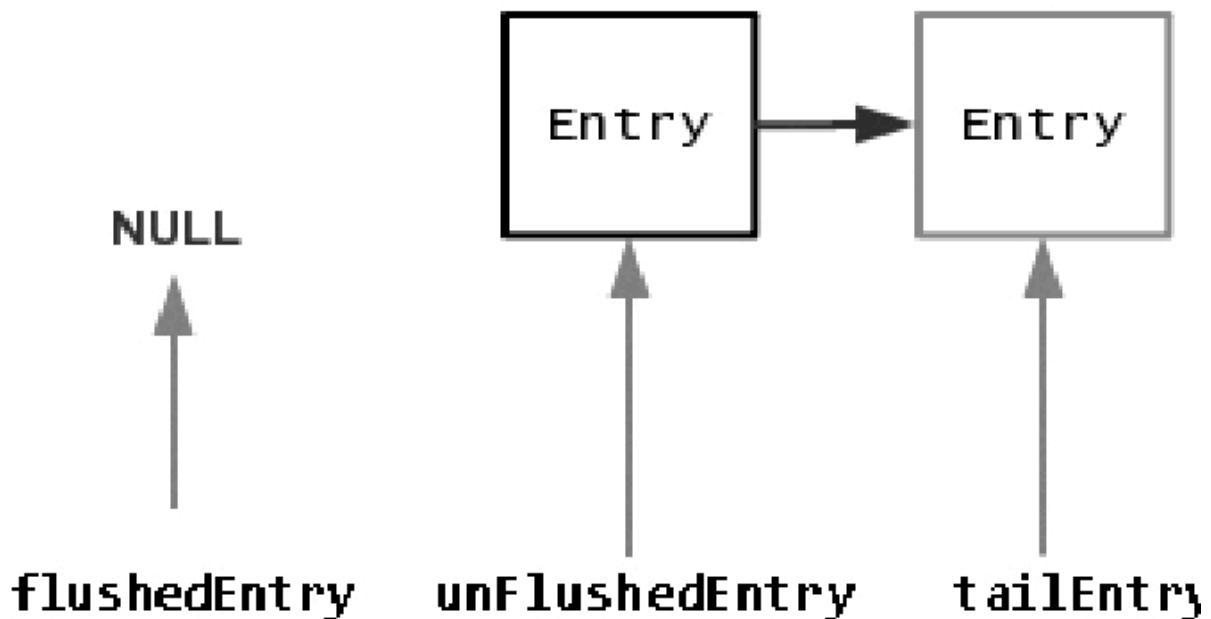
ChannelOutboundBuffer



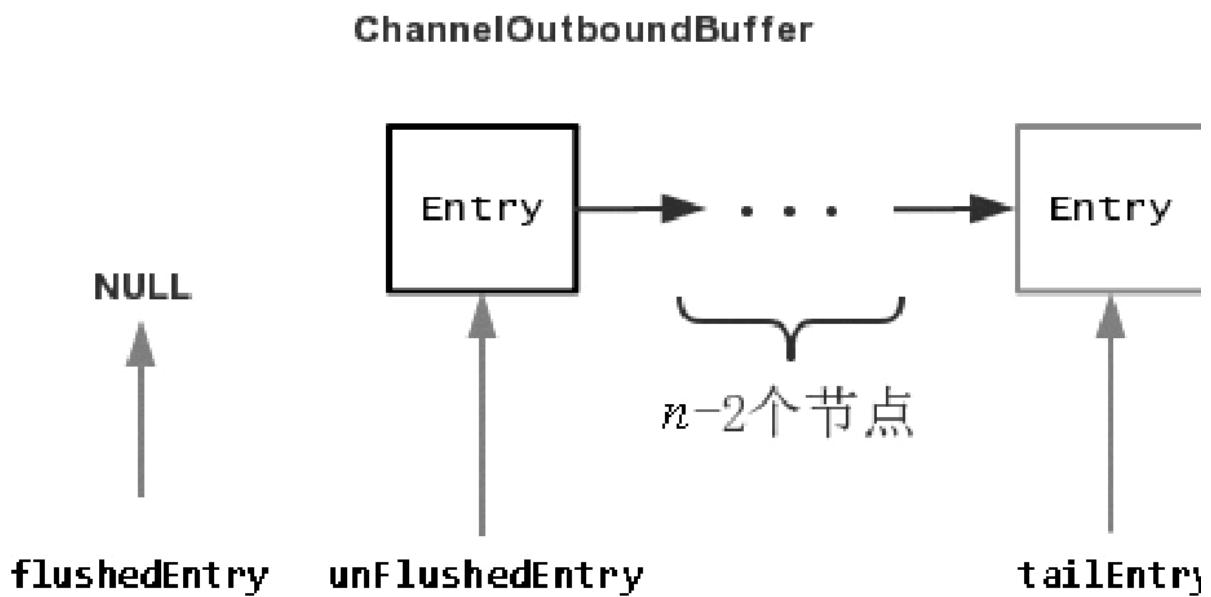
`flushedEntry`指向空，`unFlushedEntry`和`tailEntry`都指向新加入的节点。

第二次调用`addMessage`之后，各个指针的情况如下图所示。

ChannelOutboundBuffer



第 n 次调用`addMessage`之后，各个指针的情况如下图所示。



可以看到，调用 n 次`addMessage`，`flushedEntry`指针一直指向NULL，表示现在还未有节点需要写出Socket缓冲区，而`unFlushedEntry`之后有 n 个节点，表示当前还有 n 个节点尚未写出Socket缓冲区。

26.4 flush：刷新写队列

不管调用channel.flush()，还是调用ctx.flush()，最终都会落地到Pipeline中的head节点。

HeadContext

```
@Override
```

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

之后进入AbstractUnsafe。

AbstractUnsafe

```
public final void flush() {
    assertEventLoop();
    ChannelOutboundBuffer outboundBuffer = this.outboundBuffer;
    if (outboundBuffer == null) {
        return;
    }
    outboundBuffer.addFlush();
    flush0();
}
```

在flush方法中，先调用addFlush()方法。

ChannelOutboundBuffer.java

```
public void addFlush() {
```

```
Entry entry = unflushedEntry;

if (entry != null) {

if (flushedEntry == null) {

flushedEntry = entry;

}

do {

flushed ++;

if (! entry.promise.setUncancellable()) {

int pending = entry.cancel();

decrementPendingOutboundBytes(pending, false, true);

}

entry = entry.next;

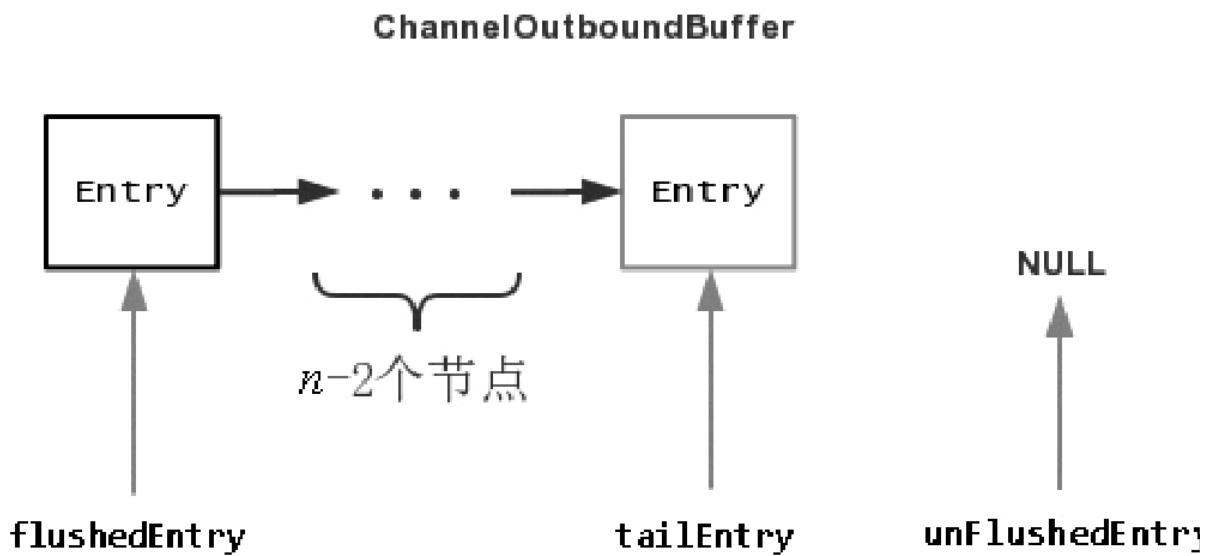
} while (entry != null);

unflushedEntry = null;

}

}
```

结合前面的图来看，首先拿到unflushedEntry指针，然后将flushedEntry指向unflushedEntry所指向的节点，调用完毕之后，三个指针的情况如下图所示。



接下来，调用flush0()。

AbstractUnsafe

```

protected void flush0() {
    doWrite(outboundBuffer);
}
  
```

发现这里的中心代码就只有一个doWrite，继续跟进。

AbstractNioByteChannel.java

```

protected void doWrite(ChannelOutboundBuffer in) throws Exception {
    int writeSpinCount = -1;
    boolean setOpWrite = false;
    for (; ; ) {
        // 获得第一个需要flush的节点的数据
        Object msg = in.current();
        if (msg instanceof ByteBuf) {
  
```

```
// 强制转换为ByteBuf，若发现没有数据可读，则直接删除该节点

ByteBuf buf = (ByteBuf) msg;

boolean done = false;

long flushedAmount = 0;

// 获得自旋锁的迭代次数

if (writeSpinCount == -1) {

    writeSpinCount = config().getWriteSpinCount();

}

// 自旋，将当前节点写出

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;

    }

}
```

```
in.progress(flushedAmount);

// 写完之后，将当前节点删除

if (done)  {

in.remove();

} else {

break;

}

}

}

}

}
```

这里略微有点复杂，我们分析一下。

26.4.1 获得第一个需要flush的节点的数据

ChannelOutBoundBuffer.java

```
public Object current() {  
    Entry entry = flushedEntry;  
  
    if (entry == null) {  
  
        return null;  
  
    }  
  
    return entry.msg;  
}
```

26.4.2 获得自旋锁的迭代次数

```
if (writeSpinCount == -1)  {

    writeSpinCount = config().getWriteSpinCount();

}
```

关于为什么要用自旋锁，Netty的文档中已经解释得很清楚，这里不再过多解释。

ChannelConfig

```
/**

* Returns the maximum loop count for a write operation until

* {@link WritableByteChannel#write(ByteBuffer)} returns a non-zero
value.

* It is similar to what a spin lock is used for in concurrency
programming.

* It improves memory utilization and write throughput depending on

* the platform that JVM runs on. The default value is {@code 16} .

*/
```

```
int getWriteSpinCount();
```

26.4.3 采用自旋方式将ByteBuf写出JDK NIO的Channel

```
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;

}

}
```

接下来，继续分析doWriteBytes方法。

```
protected int doWriteBytes(ByteBuf buf) throws Exception {

final int expectedWrittenBytes = buf.readableBytes();

return buf.readBytes(javaChannel(), expectedWrittenBytes);

}
```

我们发现，doWritesBytes方法体中出现了JavaChannel()，表明已经进入JDK NIO Channel的领域，有关Netty中ByteBuf的介绍这里不再展开介绍。

26.4.4 删除该节点

节点的数据已经写入完毕，接下来就需要删除该节点。

ChannelOutBoundBuffer.java

```
public boolean remove() {

Entry e = flushedEntry;

Object msg = e.msg;

ChannelPromise promise = e.promise;

int size = e.pendingSize;

removeEntry(e);
```

```

if (! e.cancelled) {

    ReferenceCountUtil.safeRelease(msg);

    safeSuccess(promise);

}

// 回收实体

e.recycle();

return true;

}

```

首先获得当前被flush掉的节点 (flushedEntry所指) , 然后获得该节点的回调对象 ChannelPromise, 调用removeEntry()方法移除该节点。

```

private void removeEntry(Entry e) {

    if (-- flushed == 0) {

        flushedEntry = null;

        if (e == tailEntry) {

            tailEntry = null;

            unflushedEntry = null;

        }

    }

} else {

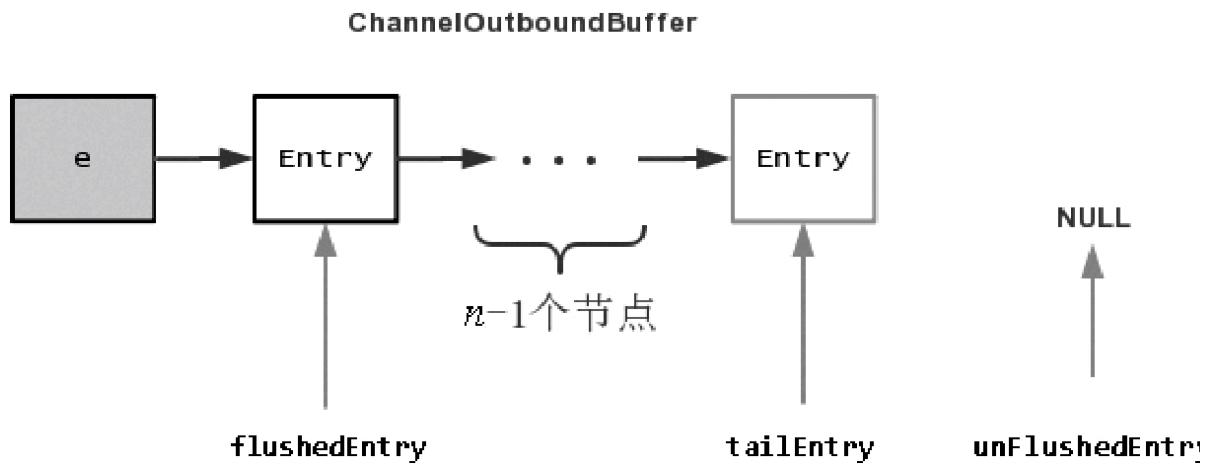
    flushedEntry = e.next;

}

}

```

这里的remove是逻辑移除，只是将flushedEntry指针移到下个节点，调用完毕之后，节点如下图所示。



随后，释放该节点数据的内存，调用safeSuccess进行回调，用户代码可以在回调里面做一些记录，下面是一段Example。

用户代码

```
ctx.write(xx).addListener(new GenericFutureListener<Future<? super Void>>() {
    @Override
    public void operationComplete(Future<? super Void> future) throws Exception {
        // 回调
    }
})
```

最后，调用recycle方法，将当前节点回收。

26.5 writeAndFlush：写队列并刷新

理解了write和flush这两个过程，writeAndFlush也就不难理解了。

writeAndFlush在某个Handler中被调用之后，最终会落到TailContext节点。

TailContext

```
public final ChannelFuture writeAndFlush(Object msg)  {

    return tail.writeAndFlush(msg);

}

public ChannelFuture writeAndFlush(Object msg)  {

    return writeAndFlush(msg, newPromise());

}

public ChannelFuture writeAndFlush(Object msg, ChannelPromise promise)
{

    write(msg, true, promise);

    return promise;

}

private void write(Object msg, boolean flush, ChannelPromise promise)
{

    AbstractChannelHandlerContext next = findContextOutbound();

    EventExecutor executor = next.executor();

    if (executor.inEventLoop())  {

        if (flush)  {

            next.invokeWriteAndFlush(m, promise);

        } else  {


```

```
next.invokeWrite(m, promise);

    }

}

}
```

可以看到，最终，通过一个boolean变量，表示是调用invokeWriteAndFlush，还是调用invokeWrite。 invokeWrite便是第26.3节中的write过程。

```
private void invokeWriteAndFlush(Object msg, ChannelPromise promise) {

    invokeWrite0(msg, promise);

    invokeFlush0();

}

}
```

可以看到，最终调用的底层方法与单独调用write和flush是一样的。

```
private void invokeWrite(Object msg, ChannelPromise promise) {

    invokeWrite0(msg, promise);

}

private void invokeFlush(Object msg, ChannelPromise promise) {

    invokeFlush0(msg, promise);

}

}
```

由此看来，invokeWriteAndFlush基本等价于write之后再来一次flush。

26.6 总结

1.Pipeline中的编码器原理是创建一个ByteBuf，将Java对象转换为ByteBuf，然后把ByteBuf继续向前传递。

2.调用write方法并没有将数据写入Socket缓冲区，而是写入了一个单向链表的数据结构中，flush才是真正的写出。

3.writeAndFlush等价于先将数据写入Netty的缓冲区，再将Netty缓冲区中的数据写入Socket缓冲区，写的过程与并发编程类似，用自旋锁保证写成功。

4.Netty缓冲区中的ByteBuf为DirectByteBuf。

第27章

本书总结

Netty相关的知识点到这里就告一段落了，最后，我们用专门一章对本书做一下总结回顾。

27.1 Netty是什么

经过整本书的学习，我们可以了解到，Netty其实可以看作对BIO和NIO的封装，并提供良好的IO读写相关的API，另外提供了非常多的开箱即用的Handler、工具类等。

27.2 服务端和客户端的启动

Netty提供了两大启动辅助类：ServerBootstrap和Bootstrap。它们的启动参数类似，都具有以下特性。

- 1.配置IO类型，配置线程模型。
- 2.配置TCP参数、Attr属性。
- 3.配置Handler。服务端除了配置Handler，还需要配置childHandler，它是定义每个连接的处理器。

27.3 ByteBuf

我们学习了Netty对二进制数据的抽象类ByteBuf。ByteBuf底层可以细分为堆内内存和堆外内存，它的API要比JDK提供的ByteBuffer更好用。ByteBuf所有的操作其实都是基于读指针和写指针来进行操作的，把申请到的一块内存划分为可读区、可写区，另外提供了自动扩容的功能。

27.4 自定义协议拆包与编解码

通常实现客户端与服务端的通信，需要自定义协议，其实就是双方商量在字节流里面，对应位置的字节段分别表示什么含义。

实际工作中用得最多的协议就是基于长度域的协议，一个协议数据包里包含了一个长度字段。在解析的时候，首先从字节流里根据自定义协议截取出一个个数据包，

使用最多的拆包器就是LengthFieldBasedFrameDecoder，只需要给它配置一些参数，即可实现自动拆包。

拆包之后呢？我们就拿到了代表字节流区段的一个个ByteBuf，解码器的作用就是把这些ByteBuf变成一个个Java对象，这样后续的handler就可以进行相应的逻辑处理。

27.5 Handler与Pipeline

Netty对逻辑处理流的处理其实和TCP协议栈的思路非常类似，分为输入和输出，也就是Inbound和Outbound类型的Handler。Inbound类Handler的添加顺序与事件传播的顺序相同，而Outbound类Handler的添加顺序与事件传播的顺序相反，这里一定要注意。

无状态的Handler可以改造为单例模式，但是千万记得要加@ChannelHandler.Sharable注解，平行等价的Handler可以使用压缩的方式缩短事件传播路径，调用ctx.xxx()而不是ctx.channel().xxx()也可以缩短事件传播路径，不过要看应用场景。

另外，每个Handler都有自己的生命周期，Netty会在Channel或者ChannelHandler处于不同状态的情况下回调相应的方法，ChannelHandler也可以动态添加，特别适用于一次性处理的Handler，用完即删除，干干净净。

27.6 耗时操作的处理与统计

对于耗时的操作，不要直接在NIO线程里做。比如，不要在channelRead0()方法里做一些访问数据库或者网络相关的逻辑，要放到自定义线程池里去做，然后要注意这个时候，writeAndFlush()方法的执行是异步的，需要通过添加监听回调的方式来判断是否执行完毕，进而进行延时的统计。而为什么要这么做，我们在源码分析篇中，深入探讨了Netty的线程模型、ChannelPipeline，以及writeAndFlush原理，相信读者此刻应该知其然并知其所以然了。

27.7 最后的话

Netty入门的门槛高，其实是因为这方面的资料太少了，并不是因为它有多难。希望通过本书，读者能够快速编写出高质量的网络应用程序。

Copyright Copyright 2010, 2012 Adobe Systems Incorporated (<http://www.adobe.com/>), with Reserved Font Name 'Source'. License This Font Software is licensed under the SIL Open Font License, Version 1.1. This license is copied below, and is also available with a FAQ at: <http://scripts.sil.org/OFL> SIL OPEN FONT LICENSE Version 1.1 - 26 February 2007 PREAMBLE The goals of the Open Font License (OFL) are to stimulate worldwide development of collaborative font projects, to support the font creation efforts of academic and linguistic communities, and to provide a free and open framework in which fonts may be shared and improved in partnership with others. The OFL allows the licensed fonts to be used, studied, modified and redistributed freely as long as they are not sold by themselves. The fonts, including any derivative works, can be bundled, embedded, redistributed and/or sold with any software provided that any reserved names are not used by derivative works. The fonts and derivatives, however, cannot be released under any other type of license. The requirement for fonts to remain under this license does not apply to any document created using the fonts or their derivatives. DEFINITIONS "Font Software" refers to the set of files released by the Copyright Holder(s) under this license and clearly marked as such. This may include source files, build scripts and documentation. "Reserved Font Name" refers to any names specified as such after the copyright statement(s). "Original Version" refers to the collection of Font Software components as distributed by the Copyright Holder(s). "Modified Version" refers to any derivative made by adding to, deleting, or substituting " in part or in whole " any of the components of the Original Version, by changing formats or by porting the Font Software to a new environment. "Author" refers to any designer, engineer, programmer, technical writer or other person who contributed to the Font Software. PERMISSION & CONDITIONS Permission is hereby granted, free of charge, to any person obtaining a copy of the Font Software, to use, study, copy, merge, embed, modify, redistribute, and sell modified and unmodified copies of the Font Software, subject to the following conditions: 1) Neither the Font Software nor any of its individual components, in Original or Modified Versions, may be sold by itself. 2) Original or Modified

Versions of the Font Software may be bundled, redistributed and/or sold with any software, provided that each copy contains the above copyright notice and this license. These can be included either as stand-alone text files, human-readable headers or in the appropriate machine-readable metadata fields within text or binary files as long as those fields can be easily viewed by the user.

3) No Modified Version of the Font Software may use the Reserved Font Name(s) unless explicit written permission is granted by the corresponding Copyright Holder. This restriction only applies to the primary font name as presented to the users.

4) The name(s) of the Copyright Holder(s) or the Author(s) of the Font Software shall not be used to promote, endorse or advertise any Modified Version, except to acknowledge the contribution(s) of the Copyright Holder(s) and the Author(s) or with their explicit written permission.

5) The Font Software, modified or unmodified, in part or in whole, must be distributed entirely under this license, and must not be distributed under any other license. The requirement for fonts to remain under this license does not apply to any document created using the Font Software.

TERMINATION

This license becomes null and void if any of the above conditions are not met.

DISCLAIMER

THE FONT SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF COPYRIGHT, PATENT, TRADEMARK, OR OTHER RIGHT. IN NO EVENT SHALL THE COPYRIGHT HOLDER BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, INCLUDING ANY GENERAL, SPECIAL, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF THE USE OR INABILITY TO USE THE FONT SOFTWARE OR FROM OTHER DEALINGS IN THE FONT SOFTWARE.

*文中代码字体版权说明

