

课程说明

- Netty是什么？
- 为什么选择Netty，而不选择原生的NIO？
- Netty的性能为什么高？
 - Java中的IO模型
 - Reactor线程模型
- Netty快速入门
- Netty核心组件
- 详解ByteBuf

1、Netty是什么？

1.1、Netty简介

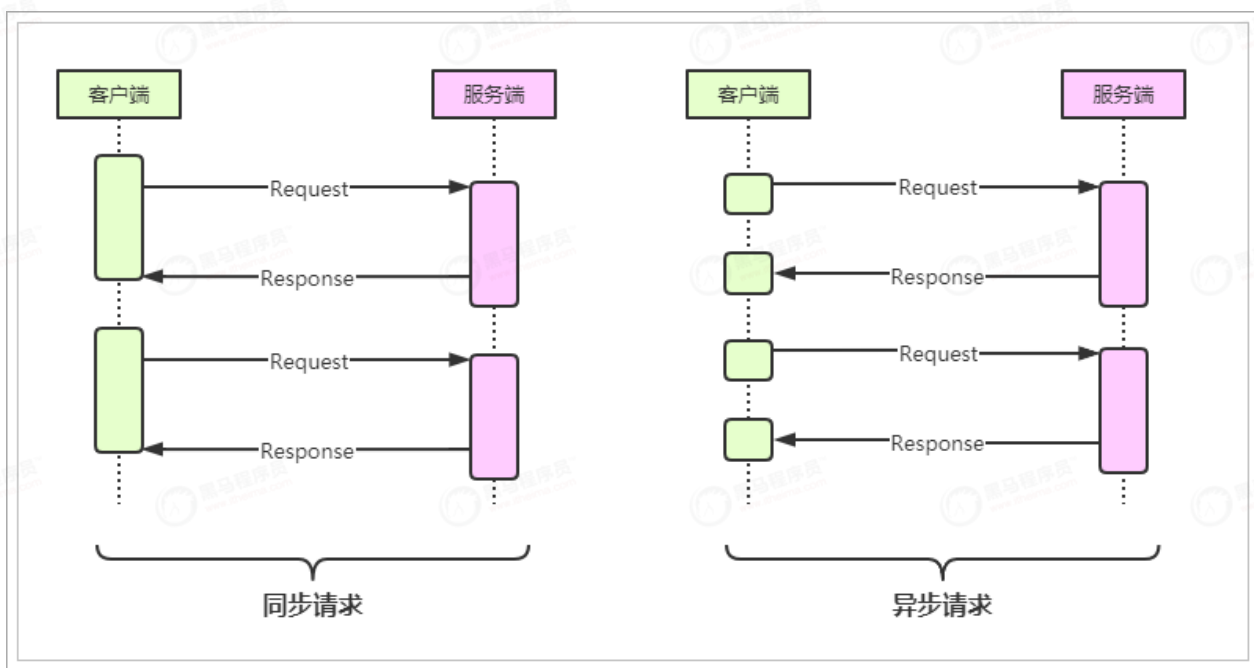
Netty是一个高性能的、异步的、基于事件驱动的网络应用框架。官网：<https://netty.io/>

官方对它有这样的描述：

Netty is an asynchronous event-driven network application framework for rapid development of maintainable high performance protocol servers & clients.

1.1.1、异步、事件驱动

同步、异步是相对的，在请求或执行过程中，如果会阻塞等待，就是同步操作，反之就是异步操作。



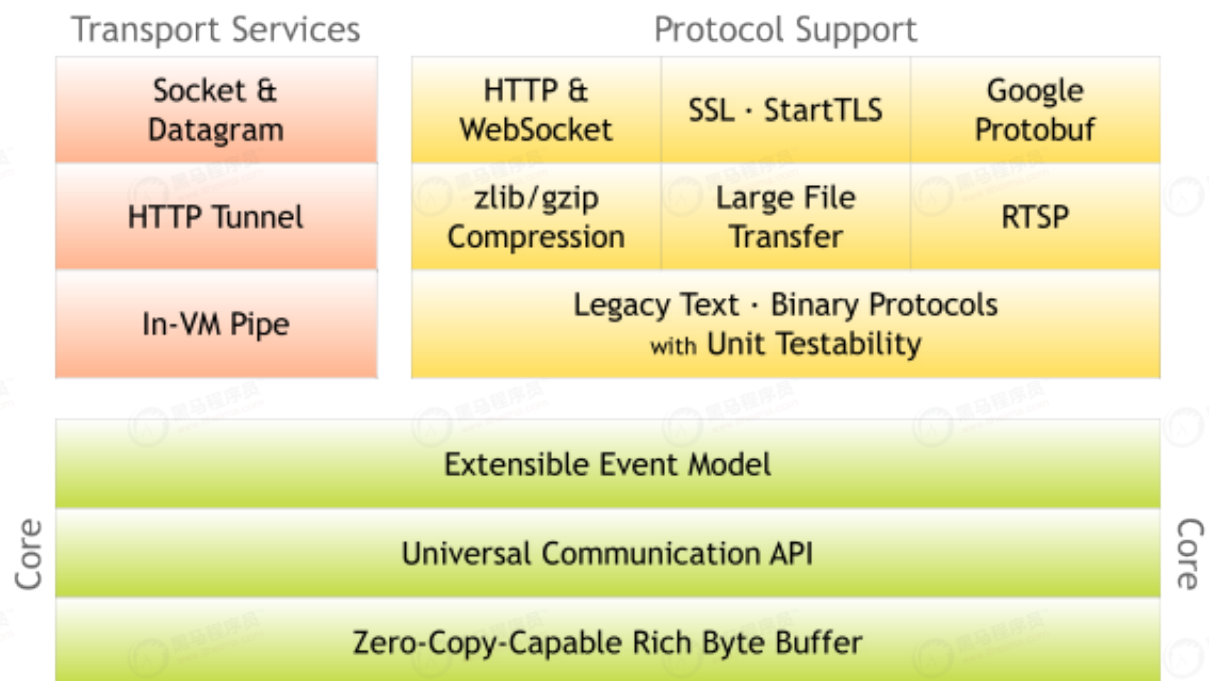
在我们熟悉的Ajax请求，就是异步并且是基于事件驱动的：

```

$(function(){
    var list = {};
    $.ajax({
        //请求方式
        type : "POST",
        //请求的媒体类型
        contentType: "application/json;charset=UTF-8",
        //请求地址
        url : "http://127.0.0.1/admin/list/",
        //数据, json字符串
        data : JSON.stringify(list),
        //请求成功
        success : function(result) {
            console.log(result);
        },
        //请求失败, 包含具体的错误信息
        error : function(e){
            console.log(e.status);
            console.log(e.responseText);
        }
    });
});

```

1.1.2、核心架构



- 核心
 - 可扩展的事件模型
 - 统一的通信api
 - 无论是http还是socket都使用统一的api, 简化了操作

- 零拷贝机制与字节缓冲区
- 传输服务
 - 支持socket以及datagram（数据报）
 - 支持http协议
 - In-VM Pipe（管道协议）
- 协议支持
 - http 以及 websocket
 - SSL 安全套接字协议支持
 - Google Protobuf（序列化框架）
 - 支持zlib、gzip压缩
 - 支持大文件的传输
 - RTSP（实时流传输协议，是TCP/IP协议体系中的一个应用层协议）
 - 支持二进制协议并且提供了完整的单元测试

1.1.3、Netty优势

- Netty是基于Java的NIO实现的，Netty将各种传输类型、协议的实现API进行了统一封装，实现了阻塞和非阻塞Socket。
- 基于事件模型实现，可以清晰的分离关注点，让开发者可以聚焦业务，提升开发效率。
- 高度可定制的线程模型-单线程、一个或多个线程池，如SEDA（Staged Event-Driven Architecture）
 - SEDA：把一个请求处理过程分成几个Stage，不同资源消耗的Stage使用不同数量的线程来处理，Stage间使用事件驱动的异步通信模式。
- Netty只依赖了JDK底层api，没有其他的依赖，如：Netty 3.X依赖JDK5以上，Netty4.x依赖JDK6以上。
- Netty在网络通信方面更加的高性能、低延迟，尽可能的减少不必要的内存拷贝，提高性能。
- 在安全方面，完整的SSL/TLS和StartTLS支持。
- 社区比较活跃，版本迭代周期短，发现bug可以快速修复，新版本也会不断的加入。

1.1.4、版本说明

Netty的版本分为，3.x、4.x和5.x，其中5.x版本已经被官方废弃，详情查看github的issue：<https://github.com/netty/netty/issues/4466>

Remove master branch #4466

Closed

normanmaurer opened this issue on 11 Nov 2015 · 28 comments

@normanmaurer

normanmaurer commented on 11 Nov 2015

Member ...

After talking with @Scottmitch and also with @nmittler I would like to propose "dropping" current master and so not release any new 5.0 version for now.

The major change of using a ForkJoinPool increases complexity and has not demonstrated a clear performance benefit. Also keeping all the branches in sync is quite some work without a real need for it as there is nothin in current master which I think justifies a new major release.

Things that we should investigate to prepare for this change:

- Deprecate `exceptionCaught` in `ChannelHandler`, only expose it in `ChannelInboundHandler`
- Expose `EventExecutorChooser` from `MultithreadEventExecutorGroup` to allow the user more flexibility to choose next `EventLoop`
- Add another method to be able to send user events both ways in the pipeline. (#4378)

Please chime in here and let me / us hear any concerns.



26



10



1

废弃5.x的主要原因是，使用ForkJoinPool后复杂度提升了，但是性能方面并没有明显的优势，反而给项目的维护带来了很大的工作量，因此还有到发布新版本的时机，所以将5.x废弃。

Netty的下载：

Netty project

News Downloads Documentation Get Involved

based legacy protocols. As a result of development, performance, stability, way to achieve ease of se.

Features

Design

- Unified API for various transport, ing socket
- Based on a flexible and extensible event model which allows clear separation of concerns

Core

4.1.50.Final - 13-May-2020

4.0.56.Final - 05-Feb-2018

3.10.6.Final - 29-Jun-2016

Apache License 2.0

Previous Releases

Nightly Builds

目前Netty的最新版本为4.1.50.Final，本套课程基于此版本学习的。

1.2、为什么选择Netty，而不选择原生的NIO？

在网络编程方面，一般都不会选择原生的NIO，而是会选择Netty、Mina等封装后的框架，主要原因是：

- NIO的类库和API繁杂，使用麻烦，需要熟练掌握Selector、ServerSocketChannel、SocketChannel、ByteBuffer等。
- 需要具备其他的额外技能做铺垫，例如熟悉Java多线程编程。这是因为NIO编程涉及到Reactor模式，你必须对多线程和网络编程非常熟悉，才能编写出高质量的NIO程序。
- 可靠性能力补齐，工作量和难度都非常大。例如客户端面临断连重连、网络闪断、半包读写、失败缓存、网络拥塞和异常码流的处理等问题，NIO编程的特点是功能开发相对容易，但是可靠性能力补齐的工作量和难度都非常大。
- JDK NIO的BUG，例如臭名昭著的epoll bug，它会导致Selector空轮询，最终导致CPU 100%。官方声称在JDK 1.6版本的update18修复了该问题，但是直到JDK 1.7版本该问题仍旧存在，只不过该BUG发生概率降低了一些而已，它并没有得到根本性解决。
 - 具体问题查看：<https://www.jianshu.com/p/3ec120ca46b2>

1.3、Netty应用场景

Netty的应用场景是非常广泛的，比如：互联网行业的、游戏行业、大数据行业、医疗行业、金融等行业。

- 互联网行业
 - 在互联网行业项目中，最具代表性的就是分布式系统架构的远程服务调用，通过RPC的方式进行高性能的服务调用，目前主流的RPC框架底层均采用了Netty作为网络通信组件。
 - 比如：阿里巴巴的分布式服务治理框架Dubbo，底层就是使用Netty作为通信组件。
 - gRPC，是Google提供的高性能RPC框架，底层也使用了Netty。
- 大数据行业
 - 大数据行业中的许多技术也采用了Netty作为通信组件，如：Flink、Spark、Elasticsearch等。

官方列出了使用Netty的一些项目：<https://netty.io/wiki/related-projects.html>

Related projects

Did you know this page is automatically generated from a Github Wiki page? You can improve it by yourself here!

Make sure the list contains only open source projects documented in English and the list is sorted alphabetically. Remove unmaintained projects (no commit activity for 12 months.)

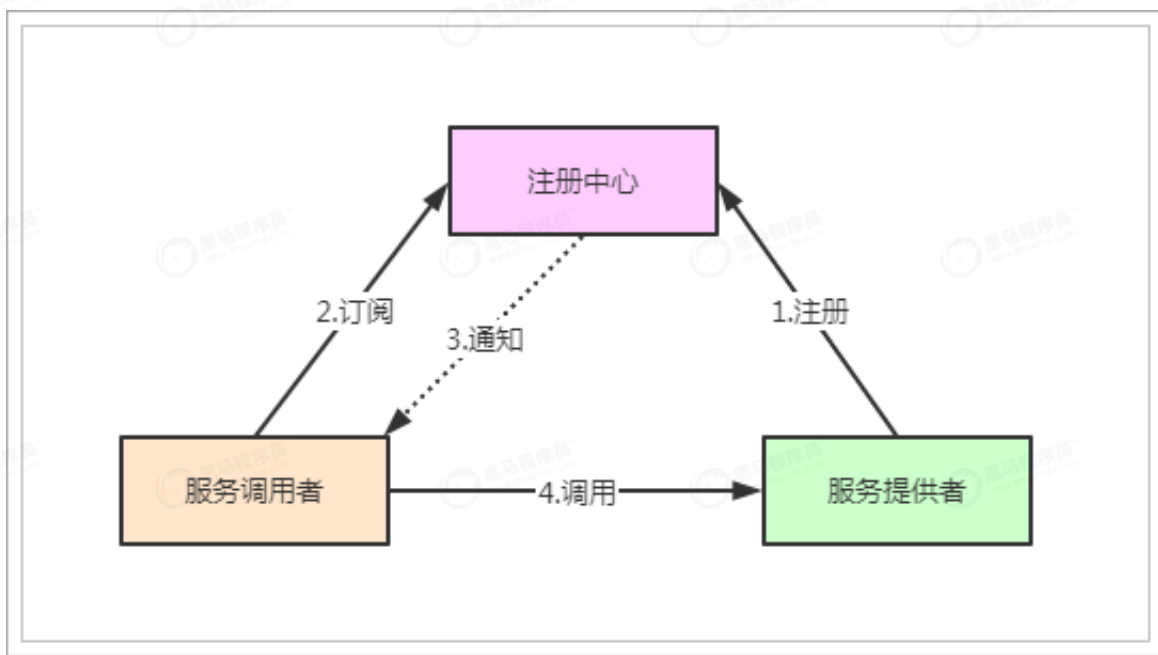
- [Akka](#) is a Scala-based platform that provides simpler scalability, fault-tolerance, concurrency, and remoting through the actor model and software transactional
- [Apache BookKeeper](#) is a scalable, fault-tolerant, and low-latency log storage.
- [Apache Cassandra](#) is a column oriented distributed database.
- [Apache Flink](#) is a distributed, stateful stream processing framework.
- [Apache James Server](#) is a modular e-mail server platform that integrates SMTP, POP3, IMAP, and NNTP.
- [Apache Pulsar](#) is an open-source distributed pub-sub messaging system.
- [Apache Spark](#) is a fast and general purpose cluster compute framework, commonly used for "Big Data" applications.
- [Apache Tajo](#) is a distributed, fault-tolerance, low-latency, and high throughput SQL engine that provides ETL features and ad-hoc query processing on large-scale data sets.
- [Arquillian](#) is an innovative in-container testing platform for the JVM
- [Async HTTP Client](#) is a simple-to-use library that allows you to execute HTTP requests and process the HTTP responses asynchronously.
- [Atomix](#) is an event-driven framework for coordinating fault-tolerant distributed systems built on the Raft consensus algorithm.
- [BungeeCord](#) is the de facto proxy solution for combining multiple Minecraft servers into a cloud / hub system.
- [Copycat](#) is a fault-tolerant state machine replication framework built on the Raft consensus algorithm.
- [Couchbase](#) is a distributed NoSQL document-oriented database that is optimized for interactive applications.
- [Elastic Search](#) is a distributed RESTful search engine built on top of Lucene.
- [Eucalyptus](#) is a software infrastructure for implementing on-premise cloud computing using an organization's own IT infrastructure, without modification, special-purpose hardware or reconfiguration.
- [Finagle](#) is an extensible RPC system for the JVM, used to construct high-concurrency servers.
- [Forest](#) is a general purpose friend-to-friend platform.
- [Gatling](#) is an asynchronous and efficient stress tool developed with Netty and Akka.
- [gRPC](#) is a high performance, open-source universal RPC framework.
- [Hammersmith](#) is a pure asynchronous MongoDB driver for Scala
- [Higgs](#) is a high performance, message oriented network library.
- [Holmes](#) is a Java application that implements DLNA/UPnP protocol for playing videos, music, pictures and podcasts (RSS) to compatible devices.
- [HornetQ](#) is a project to build a multi-protocol, embeddable, very high performance, clustered, asynchronous messaging system.
- [http-client](#) is a high performance and throughput oriented HTTP client library.
- [Infinispan](#) is an extremely scalable, highly available data grid platform.

1.4、电商系统自研RPC

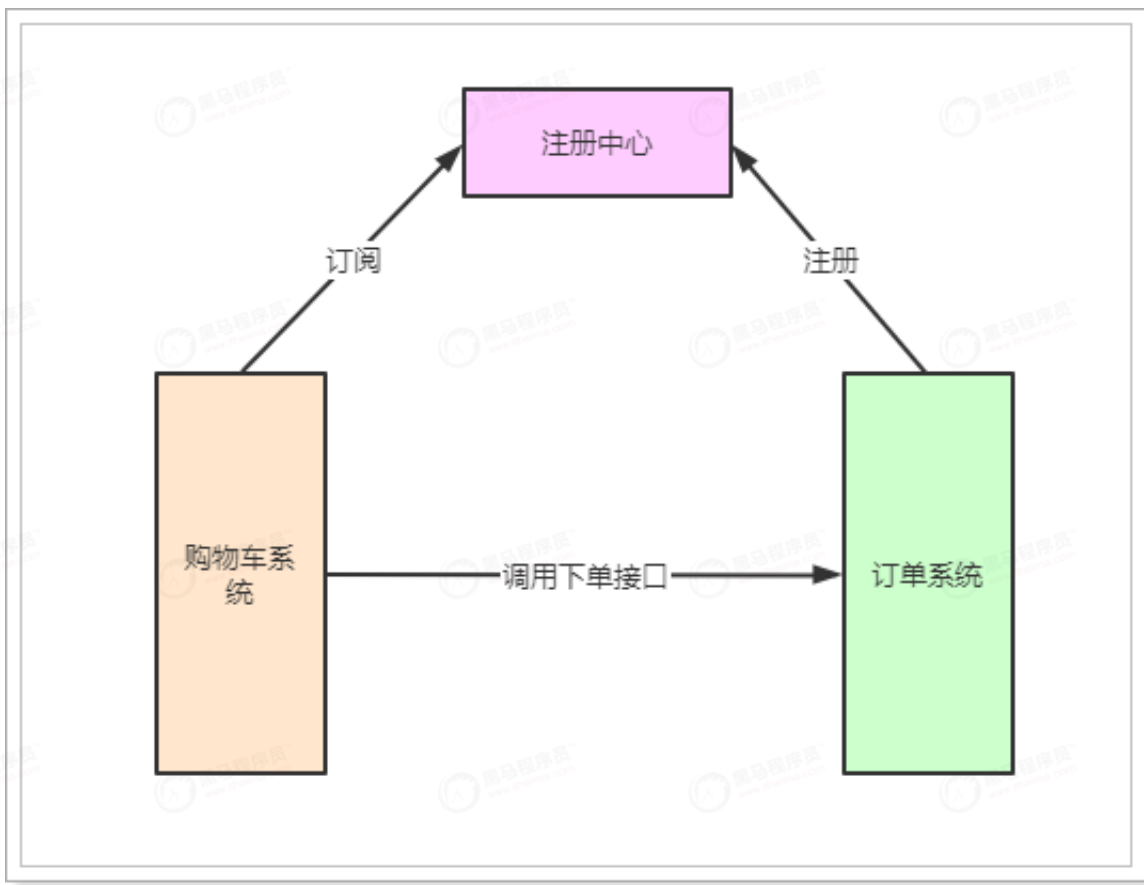
市面上有很多的RPC框架，比如：dubbo、gRPC、thrift等产品，在开发项目时，我们可以选择使用已有的RPC产品，也可以自研RPC，一线大厂一般会选择自研RPC，会根据自身的业务特点进行研发，以追求更高的性能。

本套课程，我们也从自研RPC需求出发，尝试着使用Netty来开发属于自己的RPC框架。

RPC基本的调用示意图：



在实现自研RPC后，我们将基于此来实现电商系统中的订单模块的业务，当然了，这里所实现的业务比较简单，主要是学习自研RPC为主。



2、Netty的高性能设计

在本小节中，我们来探索Netty的高性能设计，来了解Netty的性能高在哪里？

要想了解Netty的性能为什么高，就需要从Java的IO模型聊起，然后对网络编程中的Reactor线程模型理解，Netty就是使用Java的NIO实现了Reactor线程模型，理解这么多内容需要理解很多的概念，下面我们将一点点的进行学习了解。

2.1、Java中的IO模型

在JDK1.4之前，基于Java所有的socket通信都采用了同步阻塞模型（BIO），这种模型性能低下，当时大型的服务均采用C或C++开发，因为它们可以直接使用操作系统提供的异步IO或者AIO，使得性能得到大幅提升。

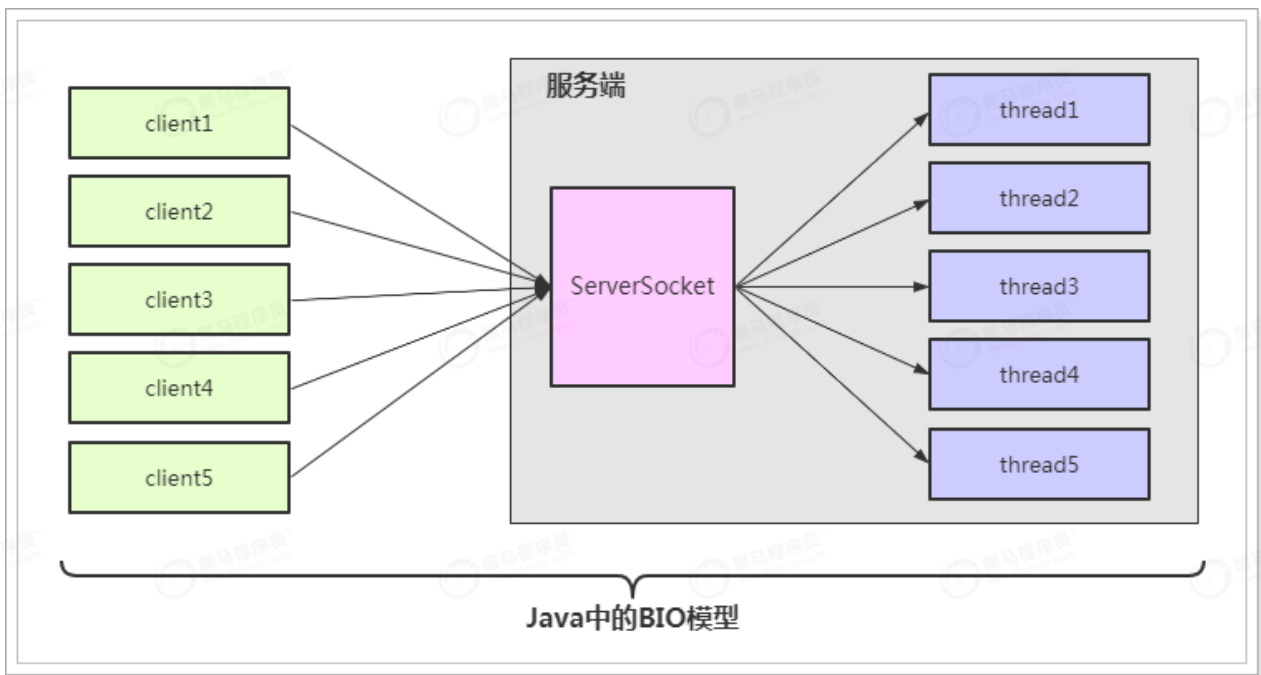
2002年，JDK1.4发布，新增了java.nio包，提供了许多异步IO开发的API和类库。新增的NIO，极大的促进了基于Java的异步非阻塞的发展和应用。

2011年，JDK7发布，将原有的NIO进行了升级，称为NIO2.0，其中也对AIO进行了支持。

2.1.1、BIO模型

java中的BIO是blocking I/O的简称，它是同步阻塞型IO，其相关的类和接口在java.io下。

BIO模型简单来讲，就是服务端为每一个请求都分配一个线程进行处理，如下：



示例代码：

```
public class BIOServer {

    public static void main(String[] args) throws Exception {

        ServerSocket serverSocket = new ServerSocket(6666);
        ExecutorService executorService = Executors.newCachedThreadPool();

        while (true) {
            System.out.println("等待客户端连接。。。");
            Socket socket = serverSocket.accept(); //阻塞
            executorService.execute(() -> {
                try {
                    InputStream inputStream = socket.getInputStream(); //阻塞
                    byte[] bytes = new byte[1024];
                    while (true){
                        int length = inputStream.read(bytes);
                        if(length == -1){
                            break;
                        }
                        System.out.println(new String(bytes, 0, length, "UTF-
8"));
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                }
            });
        }
    }
}
```

这种模式存在的问题：

- 客户端的并发数与后端的线程数成1:1的比例，线程的创建、销毁是非常消耗系统资源的，随着并发量增大，服务端性能将显著下降，甚至会发生线程堆栈溢出等错误。
- 当连接创建后，如果该线程没有操作时，会进行阻塞操作，这样极大的浪费了服务器资源。

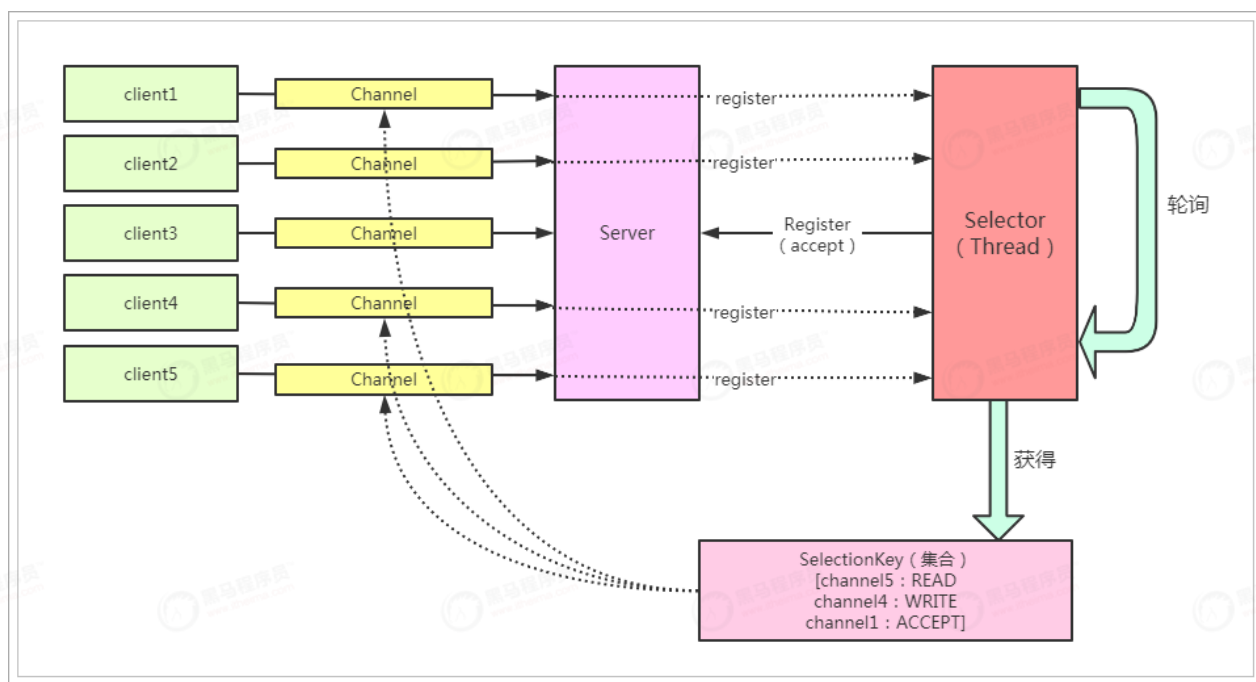
2.1.2、NIO模型

NIO，称之为New IO 或是 non-block IO（非阻塞IO），这两种说法都可以，其实称之为非阻塞IO更恰当一些。

NIO相关的代码都放在了java.nio包下，其三大核心组件：**Buffer（缓冲区）**、**Channel（通道）**、**Selector（选择器/多路复用器）**

- Buffer
 - 在NIO中，所有的读写操作都是基于缓冲区完成的，底层是通过数组实现的，常用的缓冲区是ByteBuffer，每一种java基本类型都有对应的缓冲区对象（除了Boolean类型），如：CharBuffer、IntBuffer、LongBuffer等。
- Channel
 - 在BIO中是基于Stream实现，而在NIO中是基于通道实现，与流不同的是，通道是双向的，既可以读也可以写。
- Selector
 - Selector是多路复用器，它会不断的轮询注册在其上的Channel，如果某个Channel上发生读或写事件，这个Channel就处于就绪状态，会被Selector轮询出来，然后通过SelectionKey获取就绪Channel的集合，进行IO的读写操作。

基本示意图如下：



可以看出，NIO模型要优于BIO模型，主要是：

- 通过多路复用器就可以实现一个线程处理多个通道，避免了多线程之间的上下文切换导致系统开销过大。
- NIO无需为每一个连接开一个线程处理，并且只有通道真正有有事件时，才进行读写操作，这样大

大的减少了系统开销。

示例代码：

```
public class SelectorDemo {

    /**
     * 注册事件
     *
     * @return
     */
    private Selector getSelector() throws Exception {
        //获取selector对象
        Selector selector = Selector.open();

        ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
        serverSocketChannel.configureBlocking(false); //非阻塞

        //获取通道并且绑定端口
        ServerSocket socket = serverSocketChannel.socket();
        socket.bind(new InetSocketAddress(6677));

        //注册感兴趣的事件
        serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);

        return selector;
    }

    public void listen() throws Exception {
        Selector selector = this.getSelector();

        while (true) {
            selector.select(); //该方法会阻塞，直到至少有一个事件的发生
            Set<SelectionKey> selectionKeys = selector.selectedKeys();
            Iterator<SelectionKey> iterator = selectionKeys.iterator();
            while (iterator.hasNext()) {
                SelectionKey selectionKey = iterator.next();
                process(selectionKey, selector);
                iterator.remove();
            }
        }
    }

    private void process(SelectionKey key, Selector selector) throws Exception
    {
        if(key.isAcceptable()){ //新连接请求
            ServerSocketChannel server = (ServerSocketChannel)key.channel();
            SocketChannel channel = server.accept();
            channel.configureBlocking(false); //非阻塞
        }
    }
}
```

```

        channel.register(selector, SelectionKey.OP_READ);
    }else if(key.isReadable()){ //读数据
        SocketChannel channel = (SocketChannel)key.channel();
        ByteBuffer byteBuffer = ByteBuffer.allocate(1024);
        channel.read(byteBuffer);

        System.out.println("form 客户端 " + new String(byteBuffer.array(),
0, byteBuffer.position()));
    }
}

public static void main(String[] args) throws Exception {
    new SelectorDemo().listen();
}
}

```

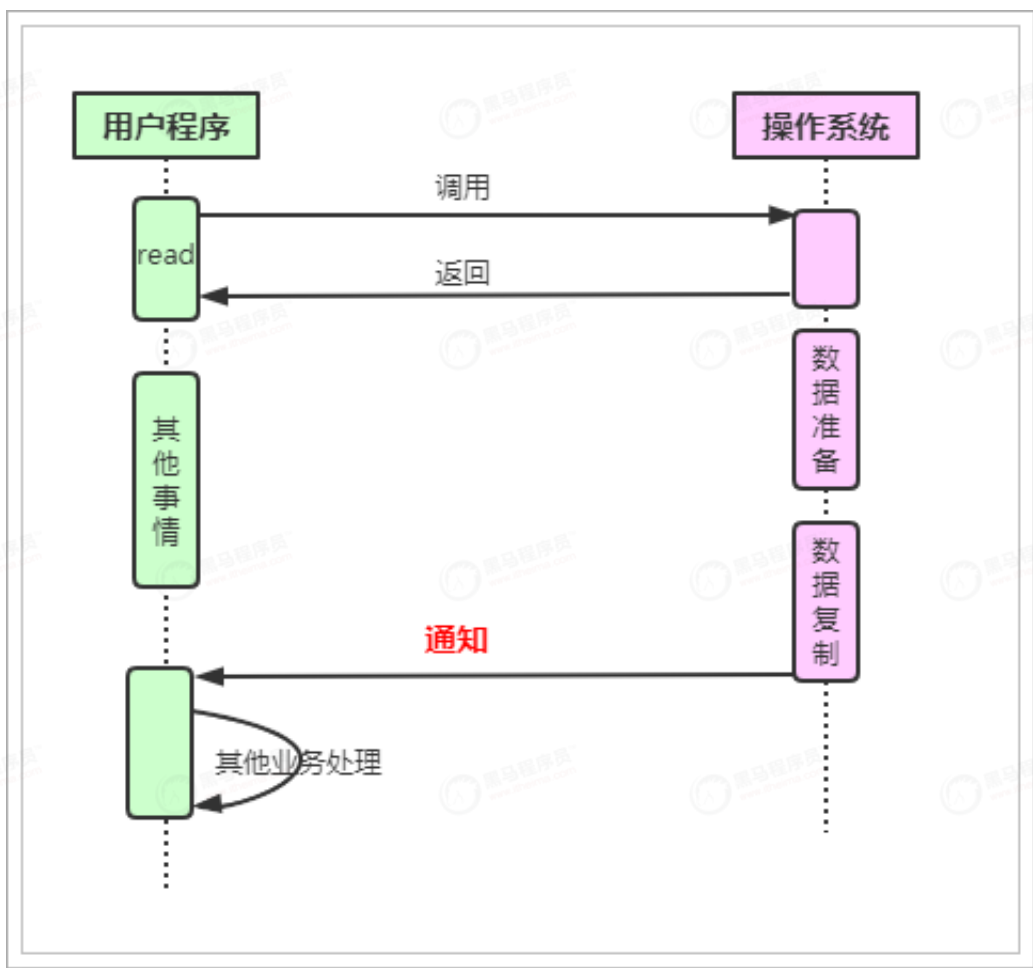
2.1.3、AIO模型

在NIO中，Selector多路复用器在做轮询时，如果没有事件发生，也会进行阻塞，如何能把这个阻塞也优化掉呢？那么AIO就在这样的背景下诞生了。

AIO是asynchronous I/O的简称，是异步IO，该异步IO是需要依赖于操作系统底层的异步IO实现。

AIO的基本流程是：用户线程通过系统调用，告知kernel内核启动某个IO操作，用户线程返回。kernel内核在整个IO操作（包括数据准备、数据复制）完成后，通知用户程序，用户执行后续的业务操作。

- kernel的数据准备
 - 将数据从网络物理设备（网卡）读取到内核缓冲区。
- kernel的数据复制
 - 将数据从内核缓冲区拷贝到用户程序空间的缓冲区。



目前AIO模型存在的不足：

- 需要完成事件的注册与传递，这里边需要底层操作系统提供大量的支持，去做大量的工作。
- Windows 系统下通过 IOCP 实现了真正的异步 I/O。但是，就目前的业界形式来说，Windows 系统，很少作为百万级以上或者说高并发应用的服务器操作系统来使用。
- 而在 Linux 系统下，异步IO模型在2.6版本才引入，目前并不完善。所以，这也是在 Linux 下，实现高并发网络编程时都是以 NIO 多路复用模型模式为主。

2.2、Reactor线程模型

Reactor线程模型不是Java专属，也不是Netty专属，它其实是一种并发编程模型，是一种思想，具有指导意义。比如，Netty就是结合了NIO的特点，应用了Reactor线程模型所实现的。

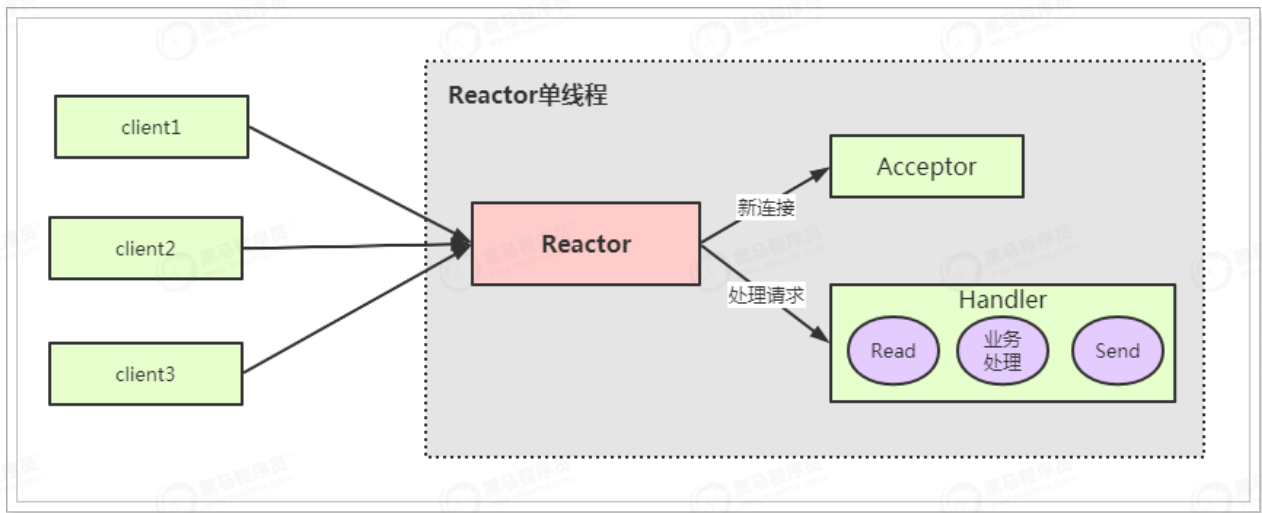
Reactor模型中定义的三种角色：

- Reactor：负责监听和分配事件，将I/O事件分派给对应的Handler。新的事件包含连接建立就绪、读就绪、写就绪等。
- Acceptor：处理客户端新连接，并分派请求到处理器链中。
- Handler：将自身与事件绑定，执行非阻塞读/写任务，完成channel的读入，完成处理业务逻辑后，负责将结果写出channel。

常见的Reactor线程模型有三种，如下：

- Reactor单线程模型
- Reactor多线程模型
- 主从Reactor多线程模型

2.2.1、单Reactor单线程模型



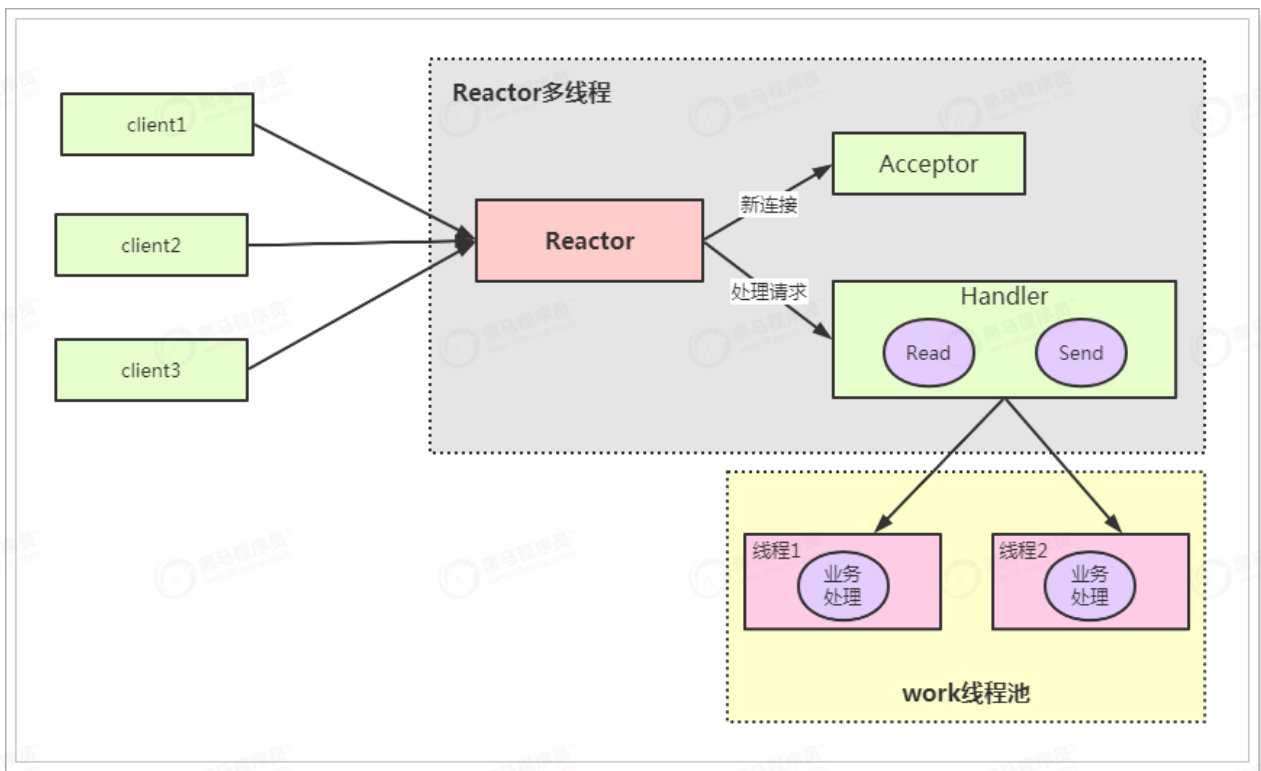
说明：

- Reactor充当多路复用器角色，监听多路连接请求，由单线程完成
- Reactor收到客户端发来的请求时，如果是新建连接通过Acceptor完成，其他的请求由Handler完成。
- Handler完成业务逻辑的处理，基本的流程是：Read --> 业务处理 --> Send。

这种模型的优缺点：

- 优点
 - 结构简单，由单线程完成，没有多线程、进程通信等问题。
 - 适合用在一些业务逻辑比较简单、对于性能要求不高的应用场景。
- 缺点
 - 由于是单线程操作，不能充分发挥多核CPU的性能。
 - 当Reactor线程负载过重之后，处理速度将变慢，这会导致大量客户端连接超时，超时之后往往会进行重发，这更加重Reactor线程的负载，最终会导致大量消息积压和处理超时，成为系统的性能瓶颈。
 - 可靠性差，如果该线程进入死循环或意外终止，就会导致整个通信系统不可用，容易造成单点故障。

2.2.2、单Reactor多线程模型



说明：

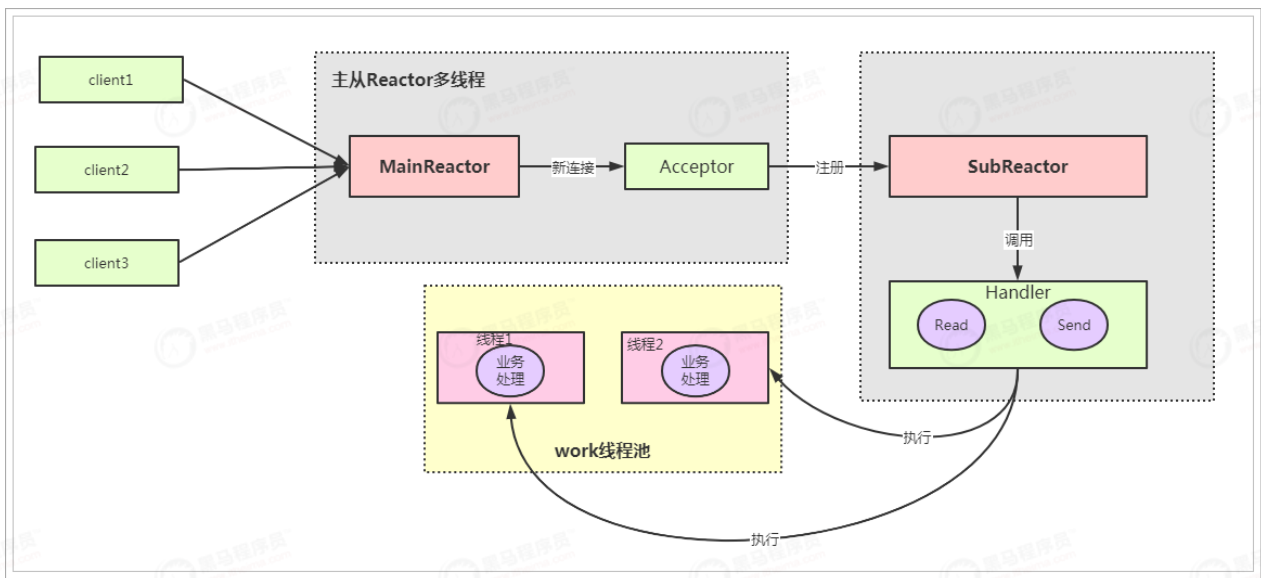
- 在Reactor多线程模型相比较单线程模型而言，不同点在于，Handler不会处理业务逻辑，只是负责响应用户请求，真正的业务逻辑，在另外的线程中完成。
- 这样可以降低Reactor的性能开销，充分利用CPU资源，从而更专注的做事件分发工作了，提升整个应用的吞吐。

但是这个模型存在的问题：

- 多线程数据共享和访问比较复杂。如果子线程完成业务处理后，把结果传递给主线程Reactor进行发送，就会涉及共享数据的互斥和保护机制。
- Reactor承担所有事件的监听和响应，只在主线程中运行，可能会存在性能问题。例如并发百万客户端连接，或者服务端需要对客户端握手进行安全认证，但是认证本身非常损耗性能。

为了解决性能问题，产生了第三种主从Reactor多线程模型。

2.2.3、主从Reactor多线程模型



在主从模型中，将Reactor分成2部分：

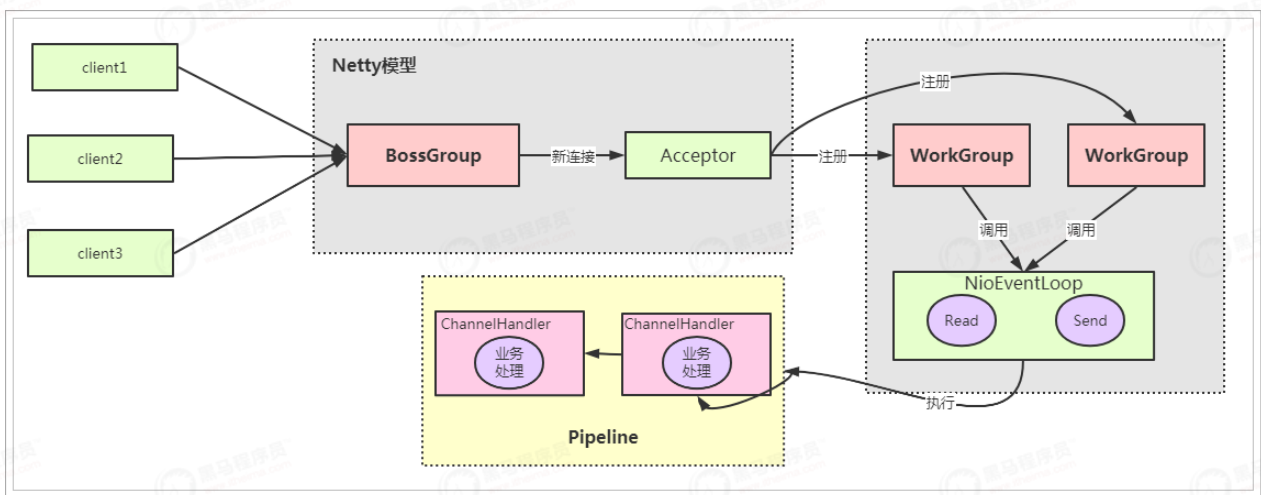
- MainReactor负责监听server socket，用来处理网络IO连接建立操作，将建立的socketChannel指定注册给SubReactor。
- SubReactor主要完成和建立起来的socket的数据交互和事件业务处理操作。

该模型的优点：

- 响应快，不必为单个同步事件所阻塞，虽然Reactor本身依然是同步的。
- 可扩展性强，可以方便地通过增加SubReactor实例个数来充分利用CPU资源。
- 可复用性高，Reactor模型本身与具体事件处理逻辑无关，具有很高的复用性。

2.3、Netty模型

Netty模型是基于Reactor模型实现的，对于以上三种模型都有非常好的支持，也非常的灵活，一般情况，在服务端会采用主从架构模型，基本示意图如下：



说明：

- 在Netty模型中，负责处理新连接事件的是BossGroup，负责处理其他事件的是WorkGroup。Group就是线程池的概念。
- NioEventLoop表示一个不断循环的执行处理任务的线程，用于监听绑定在其上的读/写事件。
- 通过Pipeline（管道）执行业务逻辑的处理，Pipeline中会有多个ChannelHandler，真正的业务逻辑

辑是在ChannelHandler中完成的。

3、Netty快速入门

开发环境：JDK8 + Idea

3.1、创建itcast-MyRPC项目

pom.xml文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>cn.itcast.myrpc</groupId>
    <artifactId>itcast-MyRPC</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
        <dependency>
            <groupId>io.netty</groupId>
            <artifactId>netty-all</artifactId>
            <version>4.1.50.Final</version>
        </dependency>

        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <!-- java编译插件 -->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.2</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                    <encoding>UTF-8</encoding>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

```
</build>
```

```
</project>
```

3.2、服务端

3.2.1、MyRPCServer

```
package cn.itcast.myrpc.server;

import cn.itcast.myrpc.server.handler.MyChannelInitializer;
import io.netty.bootstrap.ServerBootstrap;
import io.netty.channel.ChannelFuture;
import io.netty.channel.EventLoopGroup;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.nio.NioServerSocketChannel;

public class MyRPCServer {

    public void start(int port) throws Exception {
        // 主线程，不处理任何业务逻辑，只是接收客户的连接请求
        EventLoopGroup boss = new NioEventLoopGroup(1);
        // 工作线程，线程数默认是：cpu*2
        EventLoopGroup worker = new NioEventLoopGroup();

        try {
            // 服务器启动类
            ServerBootstrap serverBootstrap = new ServerBootstrap();

            serverBootstrap.group(boss, worker) //设置线程组
                .channel(NioServerSocketChannel.class) //配置server通道
                .childHandler(new MyChannelInitializer()); //worker线程的处理器

            ChannelFuture future = serverBootstrap.bind(port).sync();
            System.out.println("服务器启动完成，端口为：" + port);

            //等待服务端监听端口关闭
            future.channel().closeFuture().sync();

        } finally {
            //优雅关闭
            boss.shutdownGracefully();
            worker.shutdownGracefully();
        }
    }
}
```

3.2.2、MyChannelInitializer

```
package cn.itcast.myrpc.server.handler;

import io.netty.channel.ChannelInitializer;
import io.netty.channel.socket.SocketChannel;

/**
 * ChannelHandler的初始化
 */
public class MyChannelInitializer extends ChannelInitializer<SocketChannel> {

    @Override
    protected void initChannel(SocketChannel ch) throws Exception {
        //将业务处理器加入到列表中
        ch.pipeline().addLast(new MyChannelHandler());
    }
}
```

3.2.3、MyChannelHandler

```
package cn.itcast.myrpc.server.handler;

import io.netty.buffer.ByteBuf;
import io.netty.buffer.Unpooled;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.ChannelInboundHandlerAdapter;
import io.netty.util.CharsetUtil;

public class MyChannelHandler extends ChannelInboundHandlerAdapter {

    /**
     * 获取客户端发来的数据
     *
     * @param ctx
     * @param msg
     * @throws Exception
     */
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) throws
Exception {
        ByteBuf byteBuf = (ByteBuf) msg;
        String msgStr = byteBuf.toString(CharsetUtil.UTF_8);
        System.out.println("客户端发来数据: " + msgStr);

        //向客户端发送数据
    }
}
```

```

        ctx.writeAndFlush(Unpooled.copiedBuffer("ok", CharsetUtil.UTF_8));
    }

    /**
     * 异常处理
     *
     * @param ctx
     * @param cause
     * @throws Exception
     */
    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
    throws Exception {
        cause.printStackTrace();
        ctx.close();
    }
}

```

3.2.4、测试用例

```

package cn.itcast.myrpc;

import cn.itcast.myrpc.server.MyRPCServer;
import org.junit.Test;

public class TestServer {

    @Test
    public void testServer() throws Exception{
        MyRPCServer myRPCServer = new MyRPCServer();

        myRPCServer.start(5566);
    }

}

```

3.2.5、测试



可以看到，客户端发送数据到服务端。

3.3、客户端

3.3.1、MyRPCClient

```
package cn.itcast.myrpc.client;

import cn.itcast.myrpc.client.handler.MyClientHandler;
import io.netty.bootstrap.Bootstrap;
import io.netty.channel.ChannelFuture;
import io.netty.channel.EventLoopGroup;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.nio.NioSocketChannel;

public class MyRPCClient {

    public void start(String host, int port) throws Exception {

        //定义工作线程组
        EventLoopGroup worker = new NioEventLoopGroup();

        try {
            //注意：client使用的是Bootstrap
            Bootstrap bootstrap = new Bootstrap();
            bootstrap.group(worker)
                .channel(NioSocketChannel.class) //注意：client使用的是
                NioSocketChannel
                .handler(new MyClientHandler());

            //连接到远程服务
            ChannelFuture future = bootstrap.connect(host, port).sync();
        }
    }
}
```

```

        future.channel().closeFuture().sync();
    } finally {
        worker.shutdownGracefully();
    }
}
}

```

3.3.2、MyClientHandler

```

package cn.itcast.myrpc.client.handler;

import io.netty.buffer.ByteBuf;
import io.netty.buffer.Unpooled;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.SimpleChannelInboundHandler;
import io.netty.util.CharsetUtil;

public class MyClientHandler extends SimpleChannelInboundHandler<ByteBuf> {

    @Override
    protected void channelRead0(ChannelHandlerContext ctx, ByteBuf msg) throws
Exception {
        System.out.println("接收到服务端的消息: " +
msg.toString(CharsetUtil.UTF_8));
    }

    @Override
    public void channelActive(ChannelHandlerContext ctx) throws Exception {
        // 向服务端发送数据
        String msg = "hello";
        ctx.writeAndFlush(Unpooled.copiedBuffer(msg, CharsetUtil.UTF_8));
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
throws Exception {
        cause.printStackTrace();
        ctx.close();
    }
}

```

3.3.3、测试用例

```
package cn.itcast.myrpc;

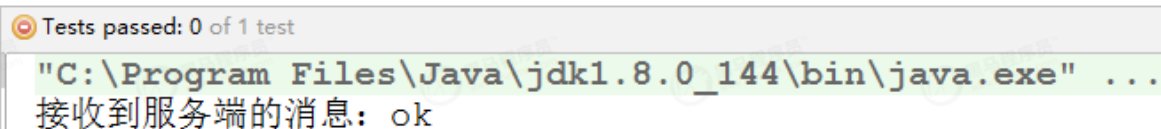
import cn.itcast.myrpc.client.MyRPCClient;
import org.junit.Test;

public class TestClient {

    @Test
    public void testClient() throws Exception{
        new MyRPCClient().start("127.0.0.1", 5566);
    }
}
```

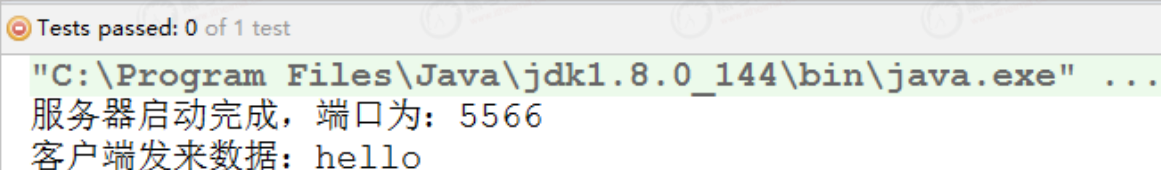
3.3.4、测试

客户端：



```
Tests passed: 0 of 1 test
"C:\Program Files\Java\jdk1.8.0_144\bin\java.exe" ...
接收到服务端的消息: ok
```

服务端：



```
Tests passed: 0 of 1 test
"C:\Program Files\Java\jdk1.8.0_144\bin\java.exe" ...
服务器启动完成, 端口为: 5566
客户端发来数据: hello
```

4、Netty核心组件

通过前面的学习，我们对Netty的整体开发有了初步的了解，在Netty中有一些核心组件，我们必须对其要有深刻的理解，下面我们一一来了解下。

4.1、Channel

Channel可以理解为是socket连接，在客户端与服务端连接的时候就会建立一个Channel，它负责基本的IO操作，比如：bind()、connect()、read()、write() 等。

Netty 的 Channel 接口所提供的 API，大大地降低了直接使用 Socket 类的复杂性。

不同协议、不同的阻塞类型的连接都有不同的 Channel 类型与之对应，常用的 Channel 类型：

- NioSocketChannel，NIO的客户端 TCP Socket 连接。
- NioServerSocketChannel，NIO的服务器端 TCP Socket 连接。
- NioDatagramChannel，UDP 连接。
- NioSctpChannel，客户端 Sctp 连接。
- NioSctpServerChannel，Sctp 服务器端连接，这些通道涵盖了 UDP 和 TCP 网络 IO 以及文件

IO。

4.2、EventLoop、EventLoopGroup

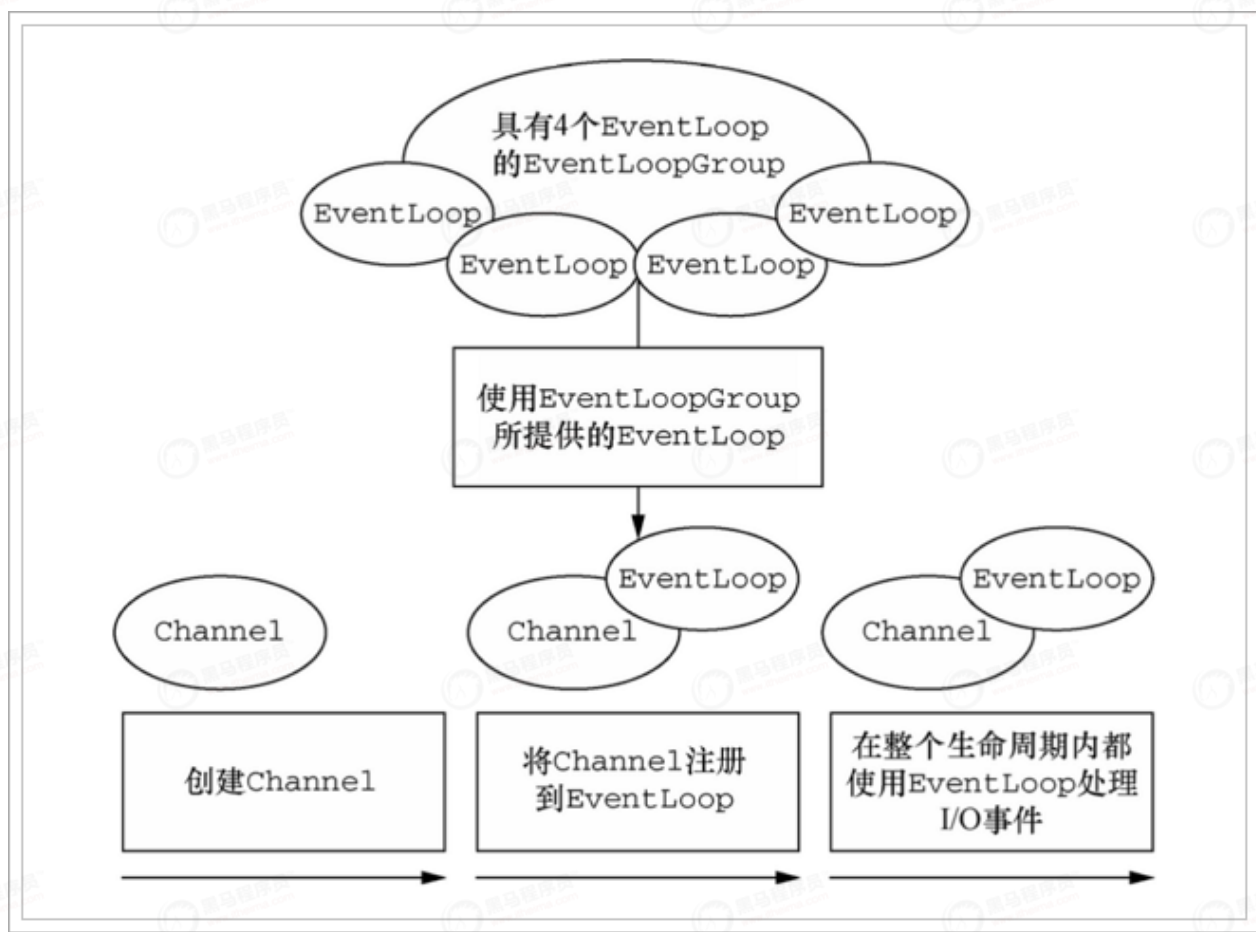
有了 Channel 连接服务，连接之间可以消息流动。如果服务器发出的消息称作“出站”消息，服务器接受的消息称作“入站”消息。那么消息的“出站”/“入站”就会产生事件（Event）。

例如：连接已激活；数据读取；用户事件；异常事件；打开链接；关闭链接等等。

有了事件，就需要一个机制去监控和协调事件，这个机制（组件）就是EventLoop。

在 Netty 中每个 Channel 都会被分配到一个 EventLoop。一个 EventLoop 可以服务于多个 Channel。

每个 EventLoop 会占用一个 Thread，同时这个 Thread 会处理 EventLoop 上面发生的所有 IO 操作和事件。



EventLoopGroup 是用来生成 EventLoop 的，在前面的例子中，第一行代码就是 `new NioEventLoopGroup();`

```
// 主线程，不处理任何业务逻辑，只是接收客户的连接请求
EventLoopGroup boss = new NioEventLoopGroup(1);
// 工作线程，线程数默认是：cpu*2
EventLoopGroup worker = new NioEventLoopGroup();
```

如果没有指定线程数大小，默认线程数为：cpu核数*2，源码如下：

```

static {
    DEFAULT_EVENT_LOOP_THREADS = Math.max(1, SystemPropertyUtil.getInt(
        "io.netty.eventLoopThreads", NettyRuntime.availableProcessors()
        * 2)); //可用cpu核数 * 2

    if (logger.isDebugEnabled()) {
        logger.debug("-Dio.netty.eventLoopThreads: {}",
            DEFAULT_EVENT_LOOP_THREADS);
    }
}

```

上图关系为：

- 一个 EventLoopGroup 包含一个或者多个 EventLoop;
- 一个 EventLoop 在它的生命周期内只和一个 Thread 绑定;
- 所有由 EventLoop 处理的 I/O 事件都将在它专有的 Thread 上被处理;
- 一个 Channel 在它的生命周期内只注册于一个 EventLoop;
- 一个 EventLoop 可能会被分配给一个或多个 Channel。

4.3、ChannelHandler

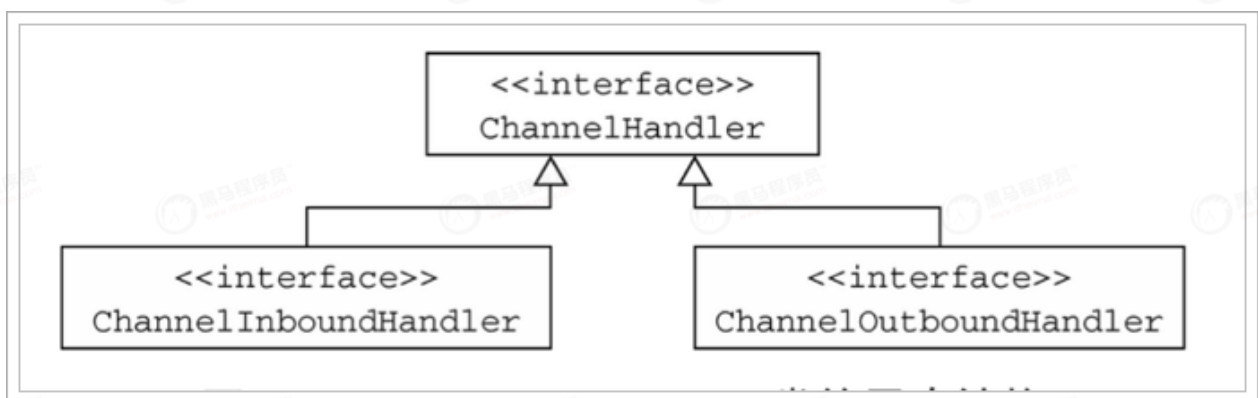
ChannelHandler对使用者而言，可以说是最重要的组件了，因为对于数据的入站和出站的业务逻辑的编写都是在ChannelHandler中完成的。

在前面的例子中，MyChannelHandler就是实现了channelRead方法，获取到客户端传来的数据。

对于数据的出站和入站，有着不同的ChannelHandler类型与之对应：

- ChannelInboundHandler 入站事件处理器
- ChannelOutboundHandler 出站事件处理器

接口继承关系如下：



ChannelHandlerAdapter提供了一些方法的默认实现，可减少用户对于ChannelHandler的编写。

ChannelInboundHandlerAdapter 与 SimpleChannelInboundHandler的区别：

- 在服务端编写ChannelHandler时继承的是ChannelInboundHandlerAdapter
- 在客户端编写ChannelHandler时继承的是SimpleChannelInboundHandler
- 两者的区别在于，前者不会释放消息数据的引用，而后者会释放消息数据的引用。

```

@Override
public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
    boolean release = true;
    try {
        if (acceptInboundMessage(msg)) {
            @SuppressWarnings("unchecked")
            I imsg = (I) msg;
            channelRead0(ctx, imsg);
        } else {
            release = false;
            ctx.fireChannelRead(msg);
        }
    } finally {
        if (autoRelease && release) {
            ReferenceCountUtil.release(msg);
        }
    }
}

```

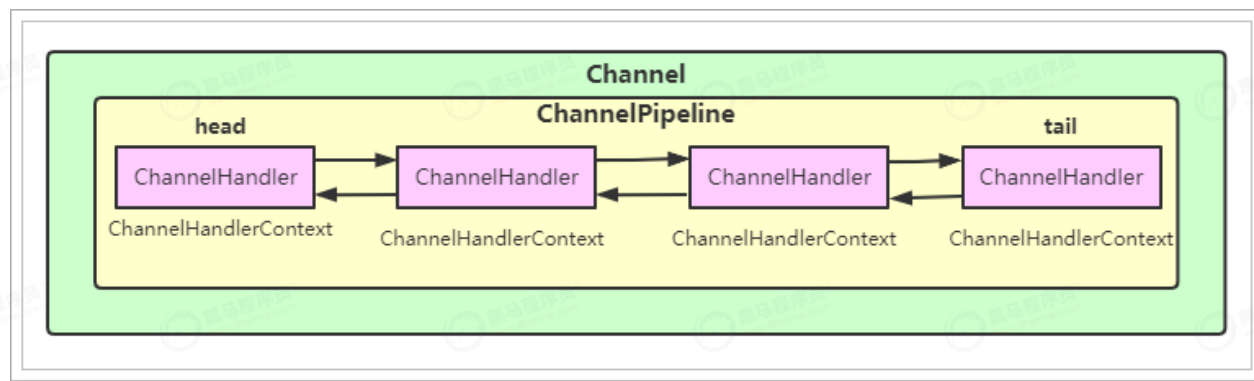
4.4、ChannelPipeline

在Channel的数据传递过程中，对应着有很多的业务逻辑需要处理，比如：编码解码处理、读写操作等，那么对于每种业务逻辑实现都需要有个ChannelHandler完成，也就意味着，一个Channel对应着多个ChannelHandler，多个ChannelHandler如何去管理它们，它们的执行顺序又该是怎么样的，这就需要ChannelPipeline进行管理了。

一个Channel包含了一个ChannelPipeline，而ChannelPipeline中维护了一个ChannelHandler的列表。

ChannelHandler与Channel和ChannelPipeline之间的映射关系，由ChannelHandlerContext进行维护。

它们关系如下：



ChannelHandler按照加入的顺序会组成一个双向链表，入站事件从链表的head往后传递到最后一个ChannelHandler，出站事件从链表的tail向前传递，直到最后一个ChannelHandler，两种类型的ChannelHandler相互不会互相影响。

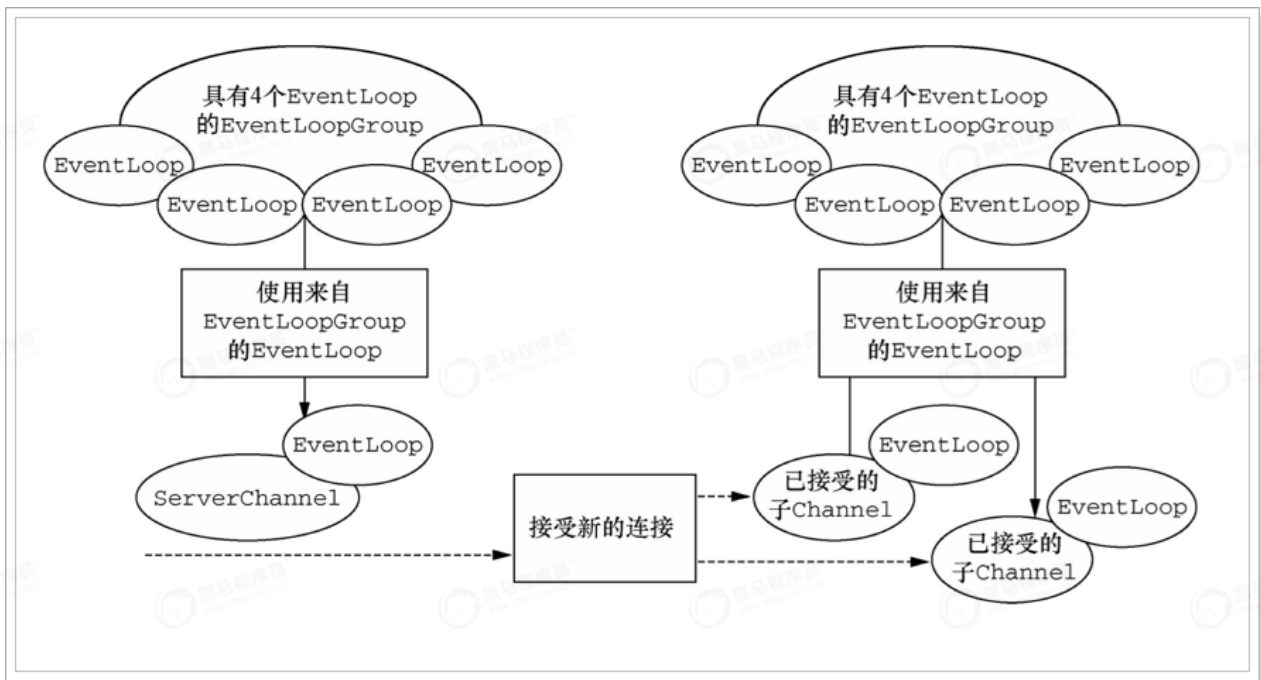
4.5、Bootstrap

Bootstrap是引导的意思，它的作用是配置整个Netty程序，将各个组件都串起来，最后绑定端口、启动Netty服务。

Netty中提供了2种类型的引导类，一种用于客户端(Bootstrap)，而另一种(ServerBootstrap)用于服务器。

它们的区别在于：

- ServerBootstrap 将绑定到一个端口，因为服务器必须要监听连接，而 Bootstrap 则是由想要连接到远程节点的客户端应用程序所使用的。
- 引导一个客户端只需要一个EventLoopGroup，但是一个ServerBootstrap则需要两个。
 - 因为服务器需要两组不同的 Channel
 - 第一组将只包含一个 ServerChannel，代表服务器自身的已绑定到某个本地端口的正在监听的套接字。
 - 第二组将包含所有已创建的用来处理传入客户端连接。



与ServerChannel相关联的EventLoopGroup 将分配一个负责为传入连接请求创建 Channel 的 EventLoop。一旦连接被接受，第二个 EventLoopGroup 就会给它的 Channel 分配一个 EventLoop。

4.6、Future

Future提供了一种在操作完成时通知应用程序的方式。这个对象可以看作是一个异步操作的结果的占位符，它将在未来的某个时刻完成，并提供对其结果的访问。

JDK 预置了 interface java.util.concurrent.Future，但是其所提供的实现，只允许手动检查对应的操作是否已经完成，或者一直阻塞直到它完成。这是非常繁琐的，所以 Netty 提供了它自己的实现——ChannelFuture，用于在执行异步操作的时候使用。

- ChannelFuture提供了几种额外的方法，这些方法使得我们能够注册一个或者多个 ChannelFutureListener实例。
- 监听器的回调方法operationComplete()，将会在对应的 操作完成时被调用 。然后监听器可以判断该操作是成功地完成了还是出错了。
- 每个 Netty 的出站 I/O 操作都将返回一个 ChannelFuture，也就是说，它们都不会阻塞。所以说，Netty完全是异步和事件驱动的。

```

    } else {
        // Registration future is almost always fulfilled already, but just in case it's not.
        final PendingRegistrationPromise promise = new PendingRegistrationPromise(channel);
        regFuture.addListener(new ChannelFutureListener() {
            @Override
            public void operationComplete(ChannelFuture future) throws Exception {
                Throwable cause = future.cause();
                if (cause != null) {
                    // Registration on the EventLoop failed so fail the ChannelPromise directly to not
                    // IllegalStateException once we try to access the EventLoop of the Channel.
                    promise.setFailure(cause);
                } else {
                    // Registration was successful, so set the correct executor to use.
                    // See https://github.com/netty/netty/issues/2586
                    promise.registered();
                }
            }
        });
        doBind0(regFuture, channel, localAddress, promise);
    }
}
return promise;
}

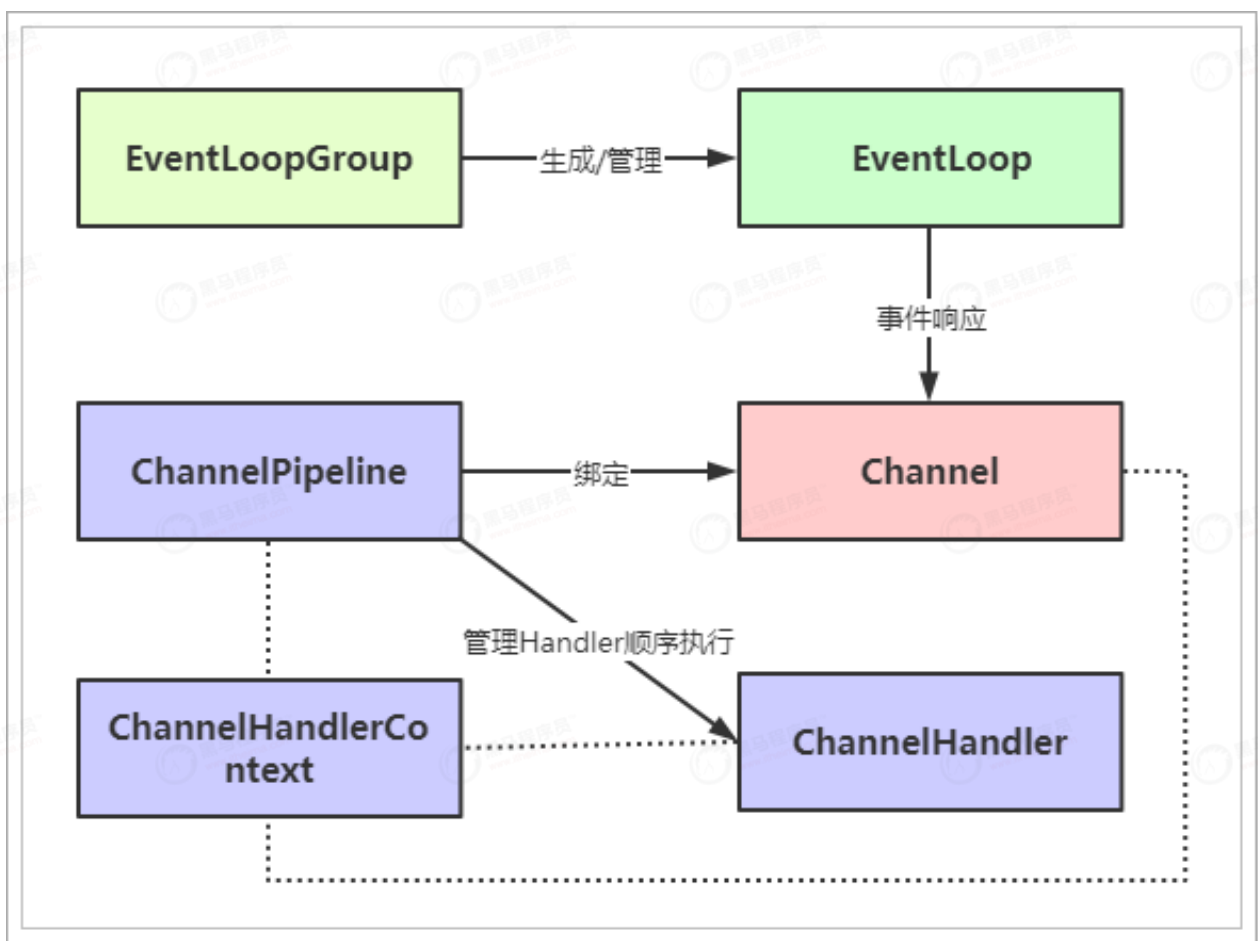
```

设置监听

操作完成的回调方法

上图是 serverBootstrap.bind(port) 方法底层的逻辑实现。

4.7、小结



通过以上图将Netty中的核心组件串起来。

5、详解ByteBuf

5.1、工作原理

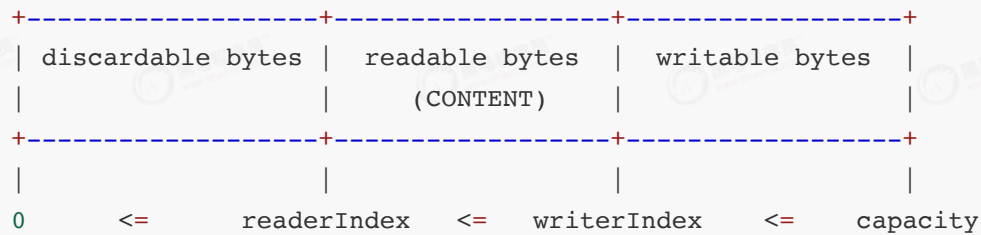
Java NIO 提供了ByteBuffer 作为它的字节容器，但是这个类使用起来过于复杂，而且也有些繁琐。Netty 的 ByteBuffer 替代品是 ByteBuf，一个强大的实现，既解决了JDK API 的局限性， 又为网络应用程序的开发者提供了更好的API。

从结构上来说，ByteBuf 由一串字节数组构成。数组中每个字节用来存放信息。

ByteBuf 提供了两个索引，一个用于读取数据，一个用于写入数据。这两个索引通过在字节数组中移动，来定位需要读或者写信息的位置。

当从ByteBuf 读取时，它的 readerIndex（读索引）将会根据读取的字节数递增。

同样，当写 ByteBuf 时，它的 writerIndex（写索引） 也会根据写入的字节数进行递增。



#discardable bytes -- 可丢弃的字节空间
#readable bytes -- 可读的字节空间
#writable bytes -- 可写的字节空间
#capacity -- 最大的容量

如果 readerIndex 超过了 writerIndex 的时候，Netty 会抛出 IndexOutOfBoundsException 异常。

5.2、基本使用

5.2.1、读取操作

```
package cn.itcast.myrpc.test;

import io.netty.buffer.ByteBuf;
import io.netty.buffer.Unpooled;
import io.netty.util.CharsetUtil;

public class TestByteBuf01 {

    public static void main(String[] args) {

        //构造
        ByteBuf byteBuf = Unpooled.copiedBuffer("hello world",
            CharsetUtil.UTF_8);

        System.out.println("byteBuf的容量为: " + byteBuf.capacity());
        System.out.println("byteBuf的可读容量为: " + byteBuf.readableBytes());
        System.out.println("byteBuf的可写容量为: " + byteBuf.writableBytes());

        while (byteBuf.isReadable()){ //方法一：内部通过移动readerIndex进行读取
```

```

        System.out.println((char)byteBuf.readByte());
    }

    //方法二：通过下标直接读取
    for (int i = 0; i < byteBuf.readableBytes(); i++) {
        System.out.println((char)byteBuf.getBytes(i));
    }

    //方法三：转化为byte[]进行读取
    byte[] bytes = byteBuf.array();
    for (byte b : bytes) {
        System.out.println((char)b);
    }
}
}

```

5.2.2、写入操作

```

package cn.itcast.myrpc.test;

import io.netty.buffer.ByteBuf;
import io.netty.buffer.Unpooled;
import io.netty.util.CharsetUtil;

public class TestByteBuf02 {

    public static void main(String[] args) {

        //构造空的字节缓冲区，初始大小为10，最大为20
        ByteBuf byteBuf = Unpooled.buffer(10, 20);

        System.out.println("byteBuf的容量为：" + byteBuf.capacity());
        System.out.println("byteBuf的可读容量为：" + byteBuf.readableBytes());
        System.out.println("byteBuf的可写容量为：" + byteBuf.writableBytes());

        for (int i = 0; i < 5; i++) {
            byteBuf.writeInt(i); //写入int类型，一个int占4个字节
        }
        System.out.println("ok");

        System.out.println("byteBuf的容量为：" + byteBuf.capacity());
        System.out.println("byteBuf的可读容量为：" + byteBuf.readableBytes());
        System.out.println("byteBuf的可写容量为：" + byteBuf.writableBytes());

        while (byteBuf.isReadable()){
            System.out.println(byteBuf.readInt());
        }
    }
}

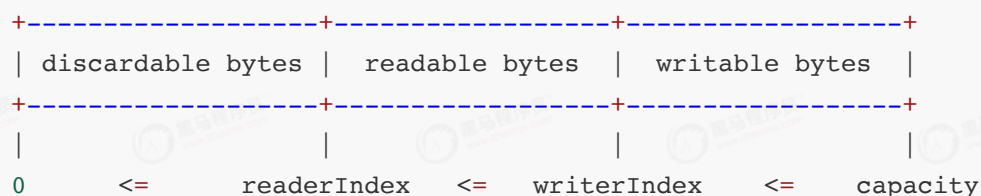
```

```
}
}
```

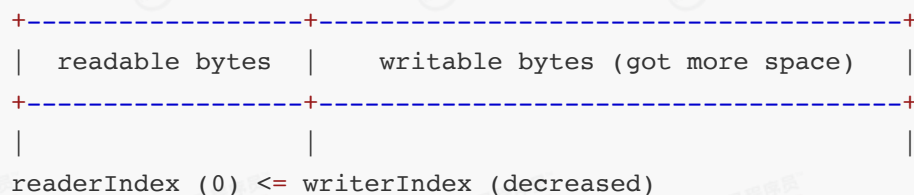
5.2.3、丢弃已读字节

#通过discardReadBytes()方可以将已经读取的数据进行丢弃处理，就可以回收已经读取的字节空间

BEFORE discardReadBytes()



AFTER discardReadBytes()



```
package cn.itcast.myrpc.test;

import io.netty.buffer.ByteBuf;
import io.netty.buffer.Unpooled;
import io.netty.util.CharsetUtil;

public class TestByteBuf03 {
    public static void main(String[] args) {

        ByteBuf byteBuf = Unpooled.copiedBuffer("hello world",
CharsetUtil.UTF_8);

        System.out.println("byteBuf的容量为: " + byteBuf.capacity());
        System.out.println("byteBuf的可读容量为: " + byteBuf.readableBytes());
        System.out.println("byteBuf的可写容量为: " + byteBuf.writableBytes());

        while (byteBuf.isReadable()){
            System.out.println((char)byteBuf.readByte());
        }

        byteBuf.discardReadBytes(); //丢弃已读的字节空间

        System.out.println("byteBuf的容量为: " + byteBuf.capacity());
    }
}
```

```

        System.out.println("byteBuf的可读容量为: " + byteBuf.readableBytes());
        System.out.println("byteBuf的可写容量为: " + byteBuf.writableBytes());
    }
}

```

5.2.4、clear()

#通过clear() 重置readerIndex、writerIndex 为0, 需要注意的是, 重置并没有删除真正的内容
BEFORE clear()

```

+-----+-----+-----+
| discardable bytes | readable bytes | writable bytes |
+-----+-----+-----+
|               |               |               |
| 0      <=    readerIndex  <=    writerIndex  <=    capacity

```

AFTER clear()

```

+-----+-----+-----+
|               | writable bytes (got more space) |
+-----+-----+-----+
|               |               |
| 0 = readerIndex = writerIndex      <=    capacity

```

```

package cn.itcast.myrpc.test;

import io.netty.buffer.ByteBuf;
import io.netty.buffer.Unpooled;
import io.netty.util.CharsetUtil;

public class TestByteBuf04 {
    public static void main(String[] args) {

        ByteBuf byteBuf = Unpooled.copiedBuffer("hello world",
        CharsetUtil.UTF_8);

        System.out.println("byteBuf的容量为: " + byteBuf.capacity());
        System.out.println("byteBuf的可读容量为: " + byteBuf.readableBytes());
        System.out.println("byteBuf的可写容量为: " + byteBuf.writableBytes());

        byteBuf.clear(); //重置readerIndex、writerIndex 为0

        System.out.println("byteBuf的容量为: " + byteBuf.capacity());
        System.out.println("byteBuf的可读容量为: " + byteBuf.readableBytes());
        System.out.println("byteBuf的可写容量为: " + byteBuf.writableBytes());
    }
}

```

```
}  
}
```

5.3、ByteBuf 使用模式

根据存放缓冲区的不同分为三类：

- **堆缓冲区 (HeapByteBuf)**，内存的分配和回收速度比较快，可以被JVM自动回收，缺点是，如果进行socket的IO读写，需要额外做一次内存复制，将堆内存对应的缓冲区复制到内核Channel中，性能会有一定程度的下降。
由于在堆上被JVM管理，在不被使用时可以快速释放。可以通过 ByteBuf.array() 来获取 byte[] 数据。
- **直接缓冲区 (DirectByteBuf)**，非堆内存，它在对外进行内存分配，相比堆内存，它的分配和回收速度会慢一些，但是将它写入或从Socket Channel中读取时，由于减少了一次内存拷贝，速度比堆内存快。
- **复合缓冲区**，顾名思义就是将上述两类缓冲区聚合在一起。Netty 提供了一个 CompositeByteBuf，可以将堆缓冲区和直接缓冲区的数据放在一起，让使用更加方便。

//默认使用的是DirectByteBuf，如果需要使用HeapByteBuf模式，则需要进行系统参数的设置

```
System.setProperty("io.netty.noUnsafe", "true"); //netty中IO操作都是基于Unsafe完成的
```

//ByteBuf 的分配要设置为非池化，否则不能切换到堆缓冲器模式

```
serverBootstrap.childOption(ChannelOption.ALLOCATOR,  
UnpooledByteBufAllocator.DEFAULT);
```

5.4、ByteBuf 的分配

Netty 提供了两种 ByteBufAllocator 的实现，分别是：

- **PooledByteBufAllocator**，实现了 ByteBuf 的对象的池化，提高性能减少并最大限度地减少内存碎片。
- **UnpooledByteBufAllocator**，没有实现对象的池化，每次会生成新的对象实例。

//通过ChannelHandlerContext获取ByteBufAllocator实例

```
ctx.alloc();
```

//通过channel也可以获取

```
channel.alloc();
```

```
//Netty默认使用了PooledByteBufAllocator

//可以在引导类中设置非池化模式
serverBootstrap.childOption(ChannelOption.ALLOCATOR,
UnpooledByteBufAllocator.DEFAULT);

//或通过系统参数设置
System.setProperty("io.netty.allocator.type", "pooled");
System.setProperty("io.netty.allocator.type", "unpooled");
```

5.5、ByteBuf的释放

ByteBuf如果采用的是堆缓冲区模式的话，可以由GC回收，但是如果采用的是直接缓冲区，就不受GC的管理，就得手动释放，否则会发生内存泄露。

关于ByteBuf的释放，分为手动释放与自动释放。

5.5.1、手动释放

手动释放，就是在使用完成后，调用ReferenceCountUtil.release(byteBuf); 进行释放。

通过release方法减去 byteBuf 的使用计数，Netty 会自动回收 byteBuf 。

示例：

```
/**
 * 获取客户端发来的数据
 *
 * @param ctx
 * @param msg
 * @throws Exception
 */
@Override
public void channelRead(ChannelHandlerContext ctx, Object msg) throws
Exception {
    ByteBuf byteBuf = (ByteBuf) msg;

    String msgStr = byteBuf.toString(CharsetUtil.UTF_8);
    System.out.println("客户端发来数据: " + msgStr);

    //释放资源
    ReferenceCountUtil.release(byteBuf);
}
```

手动释放可以达到目的，但是这种方式会比较繁琐，如果一旦忘记释放就可能会造成内存泄露。

5.5.2、自动释放

自动释放有三种方式，分别是：入站的TailHandler、继承SimpleChannelInboundHandler、HeadHandler的出站释放。

5.5.2.1、TailHandler

Netty的ChannelPipeline的流水线的末端是TailHandler，默认情况下如果每个入站处理器Handler都把消息往下传，TailHandler会释放掉ReferenceCounted类型的消息。

```
/**
 * 获取客户端发来的数据
 *
 * @param ctx
 * @param msg
 * @throws Exception
 */
@Override
public void channelRead(ChannelHandlerContext ctx, Object msg) throws
Exception {
    ByteBuf byteBuf = (ByteBuf) msg;

    String msgStr = byteBuf.toString(CharsetUtil.UTF_8);
    System.out.println("客户端发来数据: " + msgStr);

    //向客户端发送数据
    ctx.writeAndFlush(Unpooled.copiedBuffer("ok", CharsetUtil.UTF_8));

    ctx.fireChannelRead(msg); //将ByteBuf向下传递

}
```

在DefaultChannelPipeline中的TailContext内部类会在最后执行：

```
@Override
public void channelRead(ChannelHandlerContext ctx, Object msg) {
    onUnhandledInboundMessage(ctx, msg);
}

//最后会执行
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); //释放资源
    }
}
```

需要注意的是，如果没有进行向下传递，那么在TailHandler中是不会进行释放操作的。

5.5.2.2、SimpleChannelInboundHandler

当ChannelHandler继承了SimpleChannelInboundHandler后，在SimpleChannelInboundHandler的channelRead()方法中，将会进行资源的释放，我们的业务代码也需要写入到channelRead0()中。

```
//SimpleChannelInboundHandler中的channelRead()
@Override
public void channelRead(ChannelHandlerContext ctx, Object msg) throws
Exception {
    boolean release = true;
    try {
        if (acceptInboundMessage(msg)) {
            @SuppressWarnings("unchecked")
            I imsg = (I) msg;
            channelRead0(ctx, imsg);
        } else {
            release = false;
            ctx.fireChannelRead(msg);
        }
    } finally {
        if (autoRelease && release) {
            ReferenceCountUtil.release(msg); //在这里释放
        }
    }
}
```

使用：

```
package cn.itcast.myrpc.client.handler;

import io.netty.buffer.ByteBuf;
import io.netty.buffer.Unpooled;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.SimpleChannelInboundHandler;
import io.netty.util.CharsetUtil;

public class MyClientHandler extends SimpleChannelInboundHandler<ByteBuf> {

    @Override
    protected void channelRead0(ChannelHandlerContext ctx, ByteBuf msg) throws
Exception {
        System.out.println("接收到服务端的消息：" +
msg.toString(CharsetUtil.UTF_8));
    }

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

```

        // 向服务端发送数据
        String msg = "hello";
        ctx.writeAndFlush(Unpooled.copiedBuffer(msg, CharsetUtil.UTF_8));
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
        throws Exception {
        cause.printStackTrace();
        ctx.close();
    }
}

```

5.5.2.3、HeadHandler

出站处理流程中，申请分配到的 ByteBuf，通过 HeadHandler 完成自动释放。

出站处理用到的 Bytebuf 缓冲区，一般是要发送的消息，通常由应用所申请。在出站流程开始的时候，通过调用 ctx.writeAndFlush(msg)，Bytebuf 缓冲区开始进入出站处理的 pipeline 流水线。

在每一个出站Handler中的处理完成后，最后消息会来到出站的最后一棒 HeadHandler，再经过一轮复杂的调用，在flush完成后终将被release掉。

示例：

```

package cn.itcast.myrpc.client.handler;

import io.netty.buffer.ByteBuf;
import io.netty.buffer.Unpooled;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.SimpleChannelInboundHandler;
import io.netty.util.CharsetUtil;

public class MyClientHandler extends SimpleChannelInboundHandler<ByteBuf> {

    @Override
    protected void channelRead0(ChannelHandlerContext ctx, ByteBuf msg) throws
        Exception {
        System.out.println("接收到服务端的消息：" +
            msg.toString(CharsetUtil.UTF_8));
    }

    @Override
    public void channelActive(ChannelHandlerContext ctx) throws Exception {
        // 向服务端发送数据
        String msg = "hello";
        ctx.writeAndFlush(Unpooled.copiedBuffer(msg, CharsetUtil.UTF_8));
    }
}

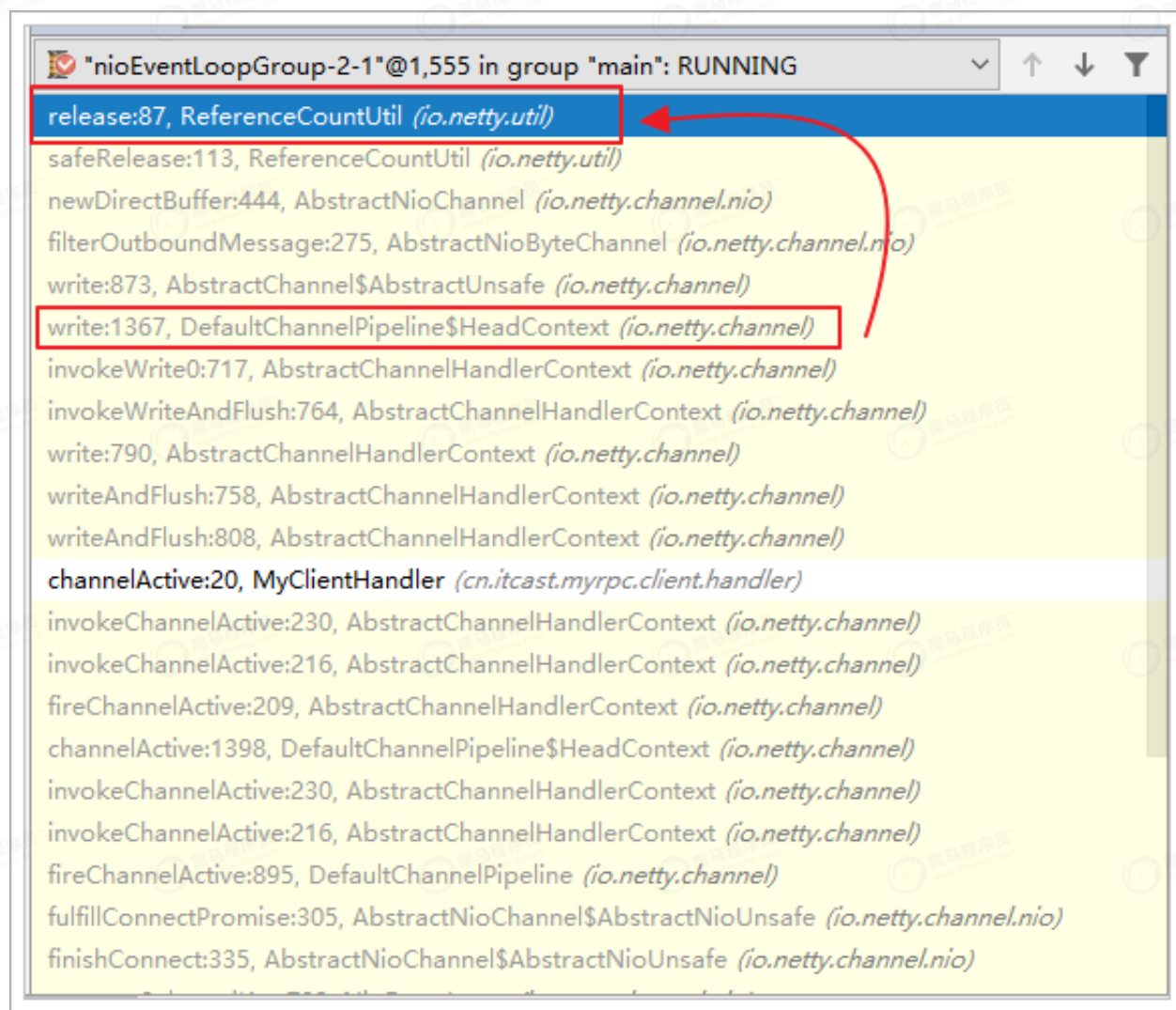
```

```

@Override
public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
throws Exception {
    cause.printStackTrace();
    ctx.close();
}
}

```

执行方法调用链：



5.5.3、小结

- 入站处理流程中，如果对原消息不做处理，调用 `ctx.fireChannelRead(msg)` 把原消息往下传，由流水线最后一棒 `TailHandler` 完成自动释放。
- 如果截断了入站处理流水线，则可以继承 `SimpleChannelInboundHandler`，完成入站 `ByteBuf` 自动释放。
- 出站处理过程中，申请分配到的 `ByteBuf`，通过 `HeadHandler` 完成自动释放。
- 入站处理中，如果将原消息转化为新的消息并调用 `ctx.fireChannelRead(newMsg)` 往下传，那必须

把原消息release掉;

- 入站处理中，如果已经不再调用 `ctx.fireChannelRead(msg)` 传递任何消息，也没有继承 `SimpleChannelInboundHandler` 完成自动释放，那更要把原消息release掉;