

Netty4.0.45 学习文档

版本号	创建人	修订日期	修订内容
V1.0	秦睿	2017-04-13	建立学习路线，完成 I/O 模型梳理
		2017-04-14	完成 Netty 简介及 HelloWorld 撰写
		2017-04-15	完成 ByteBuf、Channel 源码分析
		2017-04-16	完成数据通信及心跳检测代码
		2017-04-17	完成 EventLoop 和 ChannelPipeline 相关模块

目录

前言	4
1 常见的 I/O 模型	4
1.1 BIO、NIO 与 AIO.....	4
1.1.1 BIO.....	4
1.1.2 NIO.....	5
1.1.3 AIO.....	7
1.1.4 几种 I/O 模型总结:	7
2 Netty 简介.....	8
2.1 Netty 架构.....	8
2.2 Netty 特性.....	9
2.3 Netty 通讯步骤.....	9
3 Netty 之 HelloWorld	10
3.1 服务端.....	10
3.1.1 Server	10
3.1.2 ServerHandler	12
3.2 客户端.....	13
3.2.1 Client.....	13
3.2.2 ClientHandler	14
4 TCP 粘包/拆包问题	14
4.1 TCP 拆包和粘包	14
4.2 解决 TCP 拆包粘包的常用方法	14
4.3 Netty 粘包/拆包解决方案.....	15
5 Netty 编解码技术.....	15
5.1 Boss Marsshalling.....	15
5.2 google Protobuf.....	16
6 Netty 最佳实践.....	17
6.1 数据通信.....	17
6.2 心跳检测.....	18
7 Netty 简单源码及流程分析.....	20
7.1 ByteBuf.....	20
7.1.1 动态扩容.....	21
7.2 Channel 和 Unsafe.....	23
7.2.1 Channel	23
7.3.1 Unsafe	26
7.3 ChannelPipeline 和 ChannelHandler	30
7.3.1 ChannelPipeline	30
7.3.2 ChannelHandler.....	31
7.4 EventLoop 和 EventLoopGroup	31
7.4.1 Netty 线程模型.....	31
7.4.2 NioEventLoop.....	32
8 参考资料.....	34
9 附件	35

9.1 代码附件.....	35
10 附录(相关设计模式暂未完善)	36
10.1 涉及到的设计模式.....	36
10.1.1 Reactor 模式	36
10.1.2 Proactor 模式.....	36
10.1.3 Facade 模式	36
10.1.4 Future 模式	36
10.2 其他	36
10.2.1 Channel 配置参数	36

前言

本文档是在个人在学习 Netty 所整理的相关资料，简单的对 Netty 框架初步的介绍，暂不涉及 WebSocket、udp 协议开发、文件上传下载等，仅为方便不了解 Netty 的人少走弯路快速学习，在学习过程中如在文中发现不明确或者不正确的地方望及时指正。

1 常见的 I/O 模型

常见的几种 I/O 模型包括：同步阻塞 I/O、伪异步 I/O、非阻塞 I/O、异步 I/O。既然下面要学习的 Netty 是一个优秀的 NIO 框架，为了更好的理解 Netty 的工作原理，有必要先简单的对传统的 BIO 模型以及 NIO 模型进行一个初步的了解。

1.1 BIO、NIO 与 AIO

1.1.1 BIO

在 JDK1.4 出来之前，建立网络连接采用 BIO 模式，需要先在服务端启动 ServerSocket，然后在客户端启动 Socket 来对服务端进行通信，默认情况下服务端需要对每个请求建立一个线程等待请求，而客户端发送请求后，先咨询服务端是否有线程相应，如果没有则会一直等待或者遭到拒绝请求，如果有的话，客户端会在请求结束后继续执行。

1. BIO 模型：

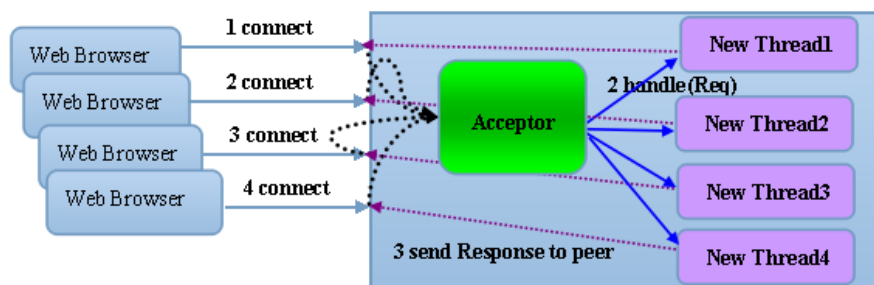


图 1 .1 BIO 通讯模型

BIO 通信模型的服务端，通常由一个独立的 Acceptor 线程负责监听客户端的连接，它接收到客户端连接请求之后为每个客户端创建一个新的线程进行链路处理，处理完成之后，通过输出流返回应答给客户端，线程销毁。这就是典型的请求一应答通信模型。

2. BIO 缺点：

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

1.1.2 NIO

1. NIO 模型

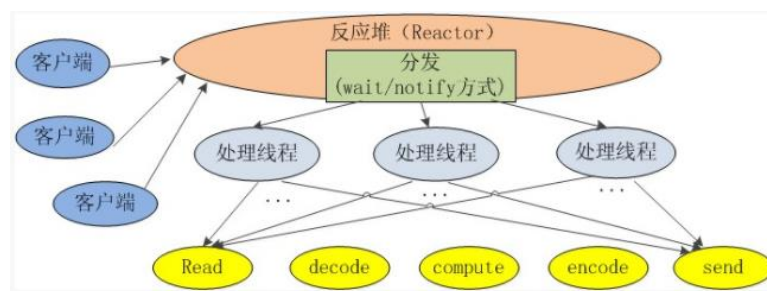


图 1.3 NIO 通讯模型

NIO 是典型的 Reactor 模型结构，每个线程的处理流程大概都是读取数据，解码，计算处理，编码，发送响应，NIO 有一个主要的类 Selector(多路复用器)，这个类似一个观察者，将需要探知的 socketChannel 注册至 Selector，当有事件发生时，传回一组 SelectionKey，读取这些 Key，就会获得刚刚注册过的 socketChannel，然后，从这个 Channel 中读取数据，接着我们可以处理这些数据。

NIO 的最重要的地方是当一个连接创建后，不需要对应一个线程，这个连接会被注册到多路复用器上面，所以所有的连接只需要一个线程就可以搞定，当这个线程中的多路复用器进行轮询的时候，发现连接上有请求的话，才开启一个线程进行处理，也就是一个请求一个线程模式。

2. NIO 相关重要概念

- **Buffer:** 所有的数据都是用缓冲区处理的，缓冲区的作用也是用来临时存储数据，可以理解为是 I/O 操作中数据的中转站。缓冲区直接为通道(Channel)服务，写入数据到通道或从通道读取数据，这样的操利用缓冲区数据来传递就可以达到对数据高效处理的目的，所有缓冲区都有 4 个属性：capacity、limit、position、mark，并遵循： $\text{capacity} \geq \text{limit} \geq \text{position} \geq \text{mark} \geq 0$ 。

属性	描述
Capacity	容量，即可以容纳的最大数据量；在缓冲区创建时被设定并且不能改变（查看源码可知当）
Limit	上界，缓冲区中当前数据量
Position	位置，下一个要被读或写的元素的索引
Mark	标记，调用 mark()来设置 mark=position，再调用 reset()可以让 position 恢复到标记的位置即 position=mark

- **Channel:** 通道，类似于流，既可以从通道中读取数据，又可以写数据到通道。但流的读写通常是单向的。通道可以异步地读写。通道中的数据总是要先读到一个 Bu

ffer, 或者总是要从一个 Buffer 中写入。

- **Selector:** 多路复用器, 提供选择已经就绪的能力, 不断轮询注册在其上的通道, 如果某个通道发生了读写操作, 这个通道就处于就绪状态, 会被 selector 轮询出来, 通过 SelectorKey 可以取得就绪的 Channel 集合, 从而进行后续 IO, 一个 Select 可以负责成千上万 Channel 通道, 没有上限, 也就是说 JDK 使用了 epoll 替代了传统的 Select 实现, 获得链接句柄没有上线, 只要一个 Selecto 轮询, 就能介入成千上万客户端。

3. NIO 序列图

a) 服务端序列图:

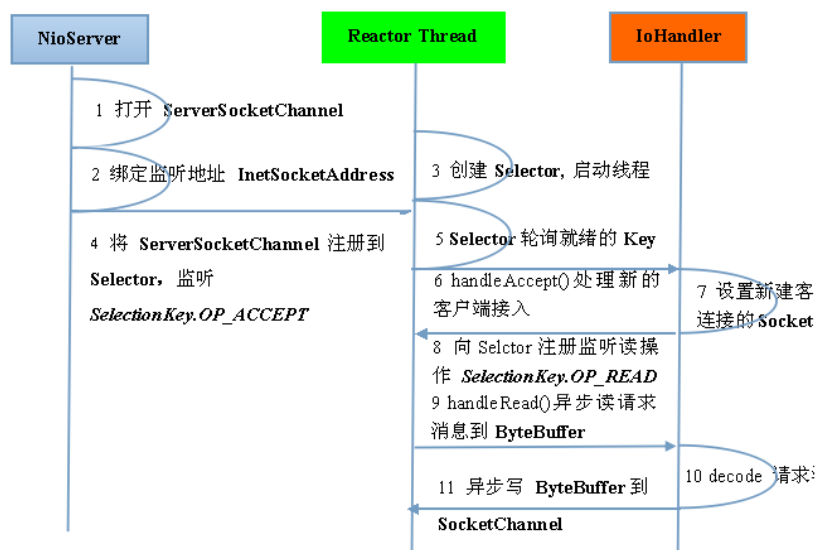


图 1.4NIO 服务端序列图

b) NIO 客户端序列图

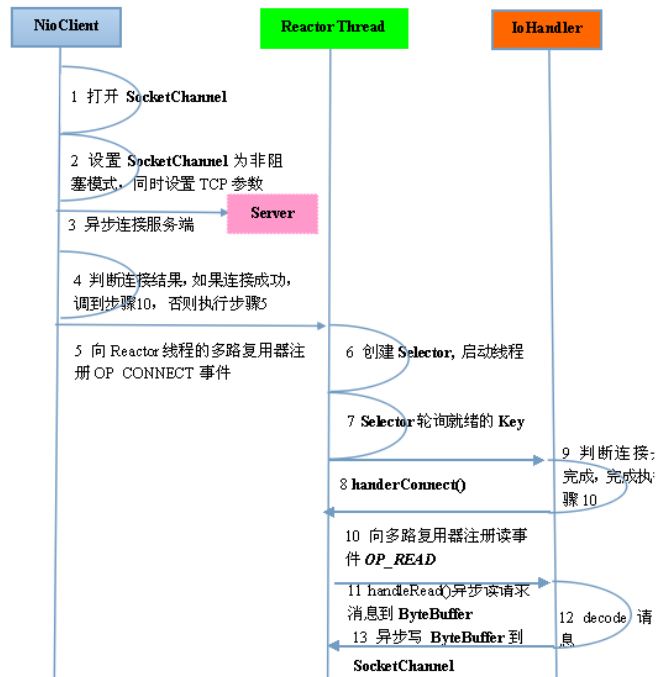


图 1.5 NIO 客户端序列图

通过上述两个序列图可以清楚的看到在 NIO 程序中程序的执行步骤。

4. JDK 的 NIO 缺点:

- NIO 类库和 API 繁杂, 使用麻烦。
- 学习成本高, 编写出高质量 NIO 程序需要知识面较广
- 存在 BUG `epoll bug` 会导致 `Selector` 空轮询, 最终导致 CPU100%。官方在 JDK1.6 版本的 `update18` 修复了该问题, 但是知道 JDK 1.7 版本该问题仍然存在, 仅仅是降低了 BUG 发生的频率, 没有从根本性解决问题。

1.1.3 AIO

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

通过 `java.util.concurrent.Future` 类来标识异步操作结果;

在执行异步操作的时候传入一个 `java.nio.channels`。

在网上看到一个介绍 AIO 的帖子, 写的比较全面, 这里就不再转述, 具体详见参考资料[15]Java aio(异步网络 IO)初探 <http://www.iteye.com/topic/472333>

1.1.4 几种 I/O 模型总结:

BIO 是一个连接一个线程。同步并阻塞, 客户端有连接请求时服务器端就需要启动一个线程进行处理, 如果这个连接不做任何事情会造成不必要的线程开销。

NIO 是一个请求一个线程。即客户端发送的连接请求都会注册到多路复用器上, 多路复用器轮询到连接有 I/O 请求时才启动一个线程进行处理。

AIO 是一个有效请求一个线程。客户端的 I/O 请求都是由 OS 先完成了再通知服务器应用去启动线程进行处理，

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

表 1.1 几种 I/O 模型的功能和特性对比

2 Netty 简介

Netty 是由 JBOSS 提供的一个 java 开源框架。Netty 提供异步的、事件驱动的网络应用程序框架和工具，用以快速开发高性能、高可靠性的网络服务器和客户端程序[错误!未找到引用源。](#)。

Netty 是一个基于 NIO 的客户、服务器端编程框架，使用 Netty 可以快速、简单的开发出网络应用。

2.1 Netty 架构

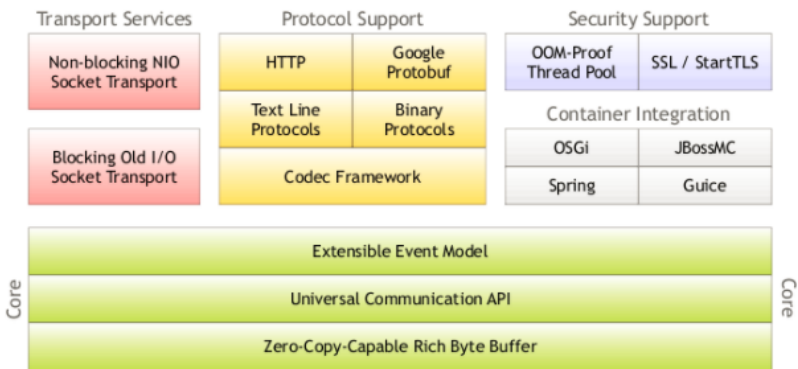


图 2.1 Netty 架构图

2.2 Netty 特性

1. 设计统一的 API，适用于不同的协议（阻塞和非阻塞）基于灵活、可扩展的事件驱动模型高度可定制的线程模型可靠的无连接数据 Socket 支持（UDP）。
2. 性能更好的吞吐量，低延迟更省资源尽量减少不必要的内存拷贝
3. 安全完整的 SSL/TLS 和 STARTTLS 的支持能在 Applet 与 Android 的限制环境运行良好
4. 健壮性不再因过快、过慢或超负载连接导致 OutOfMemoryError，不再有在高速网络环境下 NIO 读写频率不一致的问题
5. 易用完善的 JavaDoc，用户指南和样例简洁简单仅信赖于 JDK1.5

2.3 Netty 通讯步骤



图 2.2 Netty 服务端通讯步骤



图 2.3 Netty 客户端通讯步骤

- 两个 NIO 线程组，一个专门用于网络事件处理，另一个进行网络通信读写
- 创建一个 ServerBootstrap 对象 配置 Netty 的一系列参数
- 创建实际处理数据的类 ChannelInitializer，初始化准备工作，设置接受传出数据的字符集，格式，以及实际处理数据的接口
- 绑定端口，执行同步阻塞方法等待服务器端启动。

下面将结合 HelloWorld 对服务端以及客户端创建工作流程进行简单介绍。

3 Netty 之 HelloWorld

3.1 服务端

3.1.1 Server

```
public class Server {
    public static void main(String[] args) throws Exception {
        // 第一个线程组，用于接受Client端连接
        EventLoopGroup bossGroup = new NioEventLoopGroup();
        // 第二个线程组 用于实际业务的处理操作
        EventLoopGroup workerGroup = new NioEventLoopGroup();

        try {
            // 辅助类，用于配置Server
            ServerBootstrap b = new ServerBootstrap();
            // 将工作线程组加入进来
            b.group(bossGroup, workerGroup)
            // 指定NioServerSocketChannel类型的通道
            .channel(NioServerSocketChannel.class)
            // 配置日志输出
            .handler(new LoggingHandler(LogLevel.INFO))
            // 绑定具体的事件处理器
            .childHandler(new ChannelInitializer<SocketChannel>() {
                @Override
                protected void initChannel(SocketChannel ch) throws Exception {
                    ch.pipeline().addLast(new ServerHandler());
                }
            })
            .option(ChannelOption.SO_BACKLOG, 128)
            .childOption(ChannelOption.SO_KEEPALIVE, true);
            // 绑定端口，等待绑定成功
            ChannelFuture f = b.bind(12345).sync();
            // 等待服务器退出
            f.channel().closeFuture().sync();
        } finally {
            workerGroup.shutdownGracefully();
            bossGroup.shutdownGracefully();
        }
    }
}
```

在创建服务端的时候实例化了 2 个 EventLoopGroup，1 个 EventLoopGroup 实际就是一个 EventLoop 线程组，负责管理 EventLoop 的申请和释放。

查看源码可知 EventLoopGroup 管理的线程数可以通过构造函数设置，如果没有设置，默认取-Dio.netty.eventLoopThreads，如果该系统参数也没有指定，则为可用的 CPU 内核数 $\times 2$ 。

```
static {
    DEFAULT_EVENT_LOOP_THREADS = Math.max(1, SystemPropertyUtil.getInt(
        "io.netty.eventLoopThreads", Runtime.getRuntime().availableProcessors() * 2));

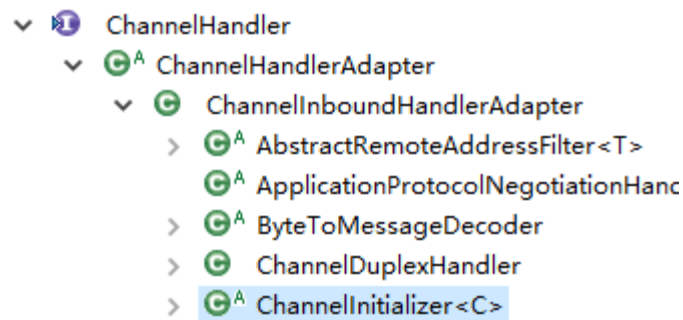
    if (logger.isDebugEnabled()) {
        logger.debug("-Dio.netty.eventLoopThreads: {}", DEFAULT_EVENT_LOOP_THREADS);
    }
}
```

bossGroup 线程组实际就是 Acceptor 线程池，负责处理客户端的 TCP 连接请求，如果系统只有一个服务端端口需要监听，则建议 bossGroup 线程组线程数设置为 1。

workerGroup 是真正负责 I/O 读写操作的线程组，通过 ServerBootstrap 的 group 方法进行设置，用于后续的 Channel 绑定。

Netty 的 NioEventLoop 读取到消息后，直接调用 ChannelPipeline 的 fireChannelRead(Object msg)。

ServerBootstrap 它是 Netty 用于启动 NIO 服务端的辅助启动类，目的是降低服务端的开发复杂度，调用 group 方法，后配置相应参数，最后绑定 I/O 事件的处理类 `new ChannelInitializer<SocketChannel>()` 它是 ChannelHandler 的实现类，类图如下



它的作用主要用于处理网络 I/O 事件，例如记录日志、对消息进行编解码等。

`ChannelFuture f = b.bind(12345).sync();` 服务端启动辅助类配置完成之后，调用它的 bind 方法绑定监听端口，随后调用它的同步阻塞方法 sync 等待绑定操作完成，完成之后 Netty 会返回一个 ChannelFuture，它的主要用于异步操作的通知回调。

使用 `f.channel().closeFuture().sync();` 方法进行阻塞，等待服务端链路关闭之后 main 函数才退出。

3.1.2 ServerHandler

```
public class ServerHandler extends ChannelInboundHandlerAdapter{

    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg)
        throws Exception {
        //将msg转换成ByteBuf对象
        ByteBuf buf = ((ByteBuf)msg);
        //通过可读字节数创建byte数组
        byte[] req = new byte[buf.readableBytes()];
        //将缓冲区的字节数组复制到新的byte数组
        buf.readBytes(req);
        String body = new String(req, "utf-8");
        System.out.println("Client:" + body);
        //创建应答消息并返回给客户端
        String resp = "服务端收到您的消息";
        ctx.writeAndFlush(Unpooled.copiedBuffer(resp.getBytes()))
        //添加监听器，关闭通道
        .addListener(ChannelFutureListener.CLOSE);
    }

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

ServerHandler 继承自 ChannelInboundHandlerAdapter，它实现了 ChannelInboundHandler 接口并继承 ChannelHandlerAdapter 的类。ChannelInboundHandlerAdapter 实现了 ChannelInboundHandler 的所有方法，作用就是处理消息并将消息转发到 ChannelPipeline 中的下一个 ChannelHandler。ChannelInboundHandlerAdapter 的 channelRead 方法处理完消息后不会自动释放消息，若想自动释放收到的消息，可以使用 SimpleChannelInboundHandler<I>。这里我们覆盖了 chanelRead() 事件处理方法。每当从客户端收到新的数据时，这个方法会在收到消息时被调用，本例子中，收到的消息的类型是 ByteBuf

3.2 客户端

3.2.1 Client

```
public class Client {  
    public static void main(String[] args) throws Exception {  
        EventLoopGroup workerGroup = new NioEventLoopGroup();  
        try {  
            Bootstrap b = new Bootstrap();  
            b.group(workerGroup);  
            b.channel(NioSocketChannel.class);  
            b.option(ChannelOption.SO_KEEPALIVE, true);  
            b.handler(new ChannelInitializer<SocketChannel>() {  
  
                @Override  
                protected void initChannel(SocketChannel ch) throws Exception {  
                    ch.pipeline().addLast(new ClientHandler());  
                }  
            });  
  
            ChannelFuture f = b.connect("127.0.0.1", 12345).sync();  
            f.channel().writeAndFlush(Unpooled.copiedBuffer("这是客户端要传输的数据".getBytes()));  
            f.channel().closeFuture().sync();  
        } finally {  
            workerGroup.shutdownGracefully();  
        }  
    }  
}
```

首先创建客户端处理 I/O 读写的 `NioEventLoopGroup()` 线程组，然后继续创建客户端辅助启动类 `Bootstrap`，随后需要对其进行配置，与服务端不同的是，它的 `Channel` 需要设置为 `b.channel(NioSocketChannel.class)`；然后添加 `Handle`。与服务器端不同的是，它的 `Channel` 需要设置为 `NioSocketChannel`，然后为其添加 `Handle`。其作用是当创建 `NioSocketChannel` 成功之后，在进行初始化时，将它的 `ChannelHandler` 设置到 `ChannelPipeline` 中，用于处理网络 I/O 事件。

客户端启用辅助类设置完成后，调用 `connect` 方法发起异步链接，然后调用同步方法等待连接成功。

最后，当客户端连接关闭后，客户端主函数退出，退出之前释放 NIO 线程组的资源。

3.2.2 ClientHandler

```
public class ClientHandler extends SimpleChannelInboundHandler<ByteBuf>{

    @Override
    protected void channelRead0(ChannelHandlerContext ctx, ByteBuf msg)
        throws Exception {
        ByteBuf buf = ((ByteBuf)msg);
        byte[] req = new byte[buf.readableBytes()];
        buf.readBytes(req);
        String body = new String(req, "utf-8");
        System.out.println("Server: " + body);
    }

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

步骤与服务端一致，接收服务端返回数据。在 Client 端我们的业务 Handler 继承的是 SimpleChannelInboundHandler，它在接收到数据后会自动 release 掉数据占用的 Bytebuffer 资源(自动调用 Bytebuffer.release())。而为何服务器端不能用呢，因为我们想让服务器把客户端请求的数据发送回去，而服务器端有可能在 channelRead 方法返回前还没有写完数据，因此不能让它自动 release。

4 TCP 粘包/拆包问题

4.1 TCP 拆包和粘包

TCP 是一个“流”协议，就是没有界限的一串数据。没有分界线。TCP 底层并不了解上层业务数据的具体含义，它会根据 TCP 缓冲区的实际情况进行包的划分，所以在业务上认为，一个完整的包可能会被 TCP 差分成多个包进行发送，也有可能把众多小包封装成一个大的数据包发送。

4.2 解决 TCP 拆包粘包的常用方法

1. 消息定长
2. 特殊字符分割
3. 将消息分为消息头和消息体，在消息头中表示消息总长度的字段

4.3 Netty 粘包/拆包解决方案

Netty 消息定长, FixedLengthFrameDecoder

```
.handler(new ChannelInitializer<SocketChannel>() {
    @Override
    protected void initChannel(SocketChannel sc) throws Exception {
        // 定长消息长度
        sc.pipeline().addLast(new FixedLengthFrameDecoder(5));
        sc.pipeline().addLast(new StringDecoder());
        sc.pipeline().addLast(new ClientHandler());
    }
});
```

Netty 特殊字符分割 DelimiterBasedFrameDecoder

```
.handler(new ChannelInitializer<SocketChannel>() {
    @Override
    protected void initChannel(SocketChannel sc) throws Exception {
        ByteBuf buf = Unpooled.copiedBuffer("$_".getBytes());
        // 定义消息最大长度和分隔符
        sc.pipeline().addLast(new DelimiterBasedFrameDecoder(1024, buf));
        // 解码器
        sc.pipeline().addLast(new StringDecoder());
        sc.pipeline().addLast(new ClientHandler());
    }
});
```




5 Netty 编解码技术

JAVA 序列化的目的主要有两个:

- 网络传输
- 对象持久化

这里重点关注网络传输。JAVA 序列化仅仅是 JAVA 编解码技术的一种, 由于它存在种种缺陷(无法跨平台、码流太大、性能低等缺点), 所以产生了较多的编解码框架。如: JBoss Marsshalling、google Protobuf、protobuf Kyro、MessagePack 等, 在跨语言情况下如性能要求不高一般可以使用 JSON 或者 XML 格式传输数据, 如性能要求较高可以使用 Thrift、MessagePack 等, 下面仅对 JBoss Marsshalling 和 google Protobuf 进行简单示例说明。

5.1 Boss Marsshalling

需要 jar 包  jboss-marshalling-1.3.0.CR9.jar  jboss-marshalling-serial-1.3.0.CR9.jar  protobuf-java-2.5.0.jar

JBoss Marsshalling 是一个 Java 对象的序列号 API 包, 修正了 JDK 自带的序列化包的很多问题, 但又保持跟 java.io.Serializable 接口兼容, 同时, 增加了一些附加属性, 可进行配置, 但

更多是在 JBoss 内部使用，应用范围有限。Marshalling 同时也支持半包和粘包的处理，只需将 Marshalling 编码器和解码器加入到 ChannelPipeline 中，就能实现对 Marshalling 序列化的支持。

1. 工具类

```
public final class MarshallingCodeCFactory {  
    /**  
     * 创建 JBoss Marshalling 解码器 MarshallingDecoder  
     * @return MarshallingDecoder  
     */  
    public static MarshallingDecoder buildMarshallingDecoder() {  
        // 首先通过 Marshalling 工具类的静态方法获取 Marshalling 实例对象 参数 serial 标识创建的是 java 序列化工厂对象。  
        final MarshallerFactory marshallerFactory = Marshalling.getProvidedMarshallerFactory("serial");  
        // 创建了 MarshallingConfiguration 对象，配置了版本号为 5  
        final MarshallingConfiguration configuration = new MarshallingConfiguration();  
        configuration.setVersion(5);  
        // 根据 MarshallerFactory 和 configuration 创建 provider  
        UnmarshallerProvider provider = new DefaultUnmarshallerProvider(marshallerFactory, configuration);  
        // 创建 Netty 的 MarshallingDecoder 对象，两个参数分别为 provider 和单个消息序列化后的最大长度  
        MarshallingDecoder decoder = new MarshallingDecoder(provider, 1024 * 1024 * 1);  
        return decoder;  
    }  
  
    /**  
     * 创建 JBoss Marshalling 编码器 MarshallingEncoder  
     * @return MarshallingEncoder  
     */  
    public static MarshallingEncoder buildMarshallingEncoder() {  
        final MarshallerFactory marshallerFactory = Marshalling.getProvidedMarshallerFactory("serial");  
        final MarshallingConfiguration configuration = new MarshallingConfiguration();  
        configuration.setVersion(5);  
        MarshallerProvider provider = new DefaultMarshallerProvider(marshallerFactory, configuration);  
        // 创建 Netty 的 MarshallingEncoder 对象，MarshallingEncoder 用于实现序列化接口的 POJO 对象序列化为二进制数组  
        MarshallingEncoder encoder = new MarshallingEncoder(provider);  
        return encoder;  
    }  
}
```

2. Client 端

```
public static void main(String[] args) throws Exception {  
    EventLoopGroup group = new NioEventLoopGroup();  
    Bootstrap b = new Bootstrap();  
    b.group(group)  
    .channel(NioSocketChannel.class)  
    .handler(new ChannelInitializer<SocketChannel>() {  
        @Override  
        protected void initChannel(SocketChannel sc) throws Exception {  
            sc.pipeline().addLast(MarshallingCodeCFactory.buildMarshallingDecoder());  
            sc.pipeline().addLast(MarshallingCodeCFactory.buildMarshallingEncoder());  
            sc.pipeline().addLast(new ClientHandler());  
        }  
    });  
    ChannelFuture cf = b.connect("127.0.0.1", 8765).sync();  
}
```

3. ServerHandler

```
@Override  
public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {  
    Request request = (Request) msg;  
    System.out.println("Server : " + request.getId() + ", " + request.getName() + ", " + request.getRequestMessage());  
}
```

*注：此处 Request 是从 client 端传入得自定义 javabean，实现 Serializable 接口

5.2 google Protobuf

google Protobuf 暂时没有进行实现。后续进行补充。

6 Netty 最佳实践

6.1 数据通信

两台机器使用 Netty 通信，大致可分为三种

1. 使用长连接通道不断开的形式进行通信，也就是服务器和客户端的通道一直处于开启状态，如果服务器的性能足够好，并且我们的客户端数量也比较少的情况下，推荐使用
2. 一次性批量提交数据，采用短连接方式。也就是我们会把数据保存在本地临时缓冲区或者临时表里，当达到临界值时进行一次性批量提交，又或者根据定时任务轮询提交，这种情况弊端是做不到实时性传输，在对实时性不高的应用程序中推荐
3. 可以使用一种特殊的长连接，在指定某一时间内，服务器与某台客户端没有任何通信，则断开连接。下次连接则是客户端向服务器发送请求的时候，再次建立连接

当我们采取第三种措施，那么会存在两种问题：a) 如何在超时后关闭通道，如何再次建立连接 获取 channel 进行判断？使用 ReadTimeoutHandler 可以控制超时时间，当通道关闭后重新获取。b) 服务器宕机时，客户端如何与服务器进行连接呢。可以使用定时任务进行轮询。下面将详细的对第一种方式进行简单描述。这里有 7 个类，分别是 Client，ClientHandler、Server、ServerHandler、Request（客户端请求的实体类）、Response（服务端返回给客户端的实体类）、MarshallingCodeCFactory（Marshalling 工厂）。Channel 核心代码如下，具体代码详见附件。

Channel 代码如下：

```
public class Client {
    //静态内部类实现单例模式
    private static class SingletonHolder {
        static final Client instance = new Client();
    }

    public static Client getInstance(){
        return SingletonHolder.instance;
    }

    private EventLoopGroup group;
    private Bootstrap b;
    private ChannelFuture cf ;

    private Client(){
        group = new NioEventLoopGroup();
        b = new Bootstrap();
        b.group(group)
        .channel(NioSocketChannel.class)
        .handler(new LoggingHandler(LogLevel.INFO))
        .handler(new ChannelInitializer<SocketChannel>() {
            @Override
            protected void initChannel(SocketChannel sc) throws Exception {
                sc.pipeline().addLast(MarshallingCodeCFactory.buildMarshallingDecoder());
                sc.pipeline().addLast(MarshallingCodeCFactory.buildMarshallingEncoder());
                //超时handler（当服务器端与客户端在指定时间以上没有任何进行通信，则会关闭响应的通道，主要为减小服务端资源占用）
                sc.pipeline().addLast(new ReadTimeoutHandler(5));
                sc.pipeline().addLast(new ClientHandler());
            }
        });
    }
}
```

下面再来看 Client 端主函数：

```

public static void main(String[] args) throws Exception{
    final Client c = Client.getInstance();

    ChannelFuture cf = c.getChannelFuture();
    for(int i = 1; i <= 3; i++){
        Request request = new Request();
        request.setId("" + i);
        request.setName("pro" + i);
        request.setRequestMessage("数据信息" + i);
        cf.channel().writeAndFlush(request);
        //休眠4秒，上面超时时间5秒故此处有数据发送不会超时，数据循环三次传输完后链接超时
        TimeUnit.SECONDS.sleep(4);
    }

    cf.channel().closeFuture().sync();
    //模拟重新建立连接
    new Thread(new Runnable() {
        @Override
        public void run() {
            try {
                System.out.println("进入子线程...");
                //重新获取ChannelFuture
                ChannelFuture cf = c.getChannelFuture(); 获取链接方法在下面进行描述
                //再次发送数据
                Request request = new Request();
                request.setId("" + 4);
                request.setName("pro" + 4);
                request.setRequestMessage("数据信息" + 4);
                cf.channel().writeAndFlush(request);
                cf.channel().closeFuture().sync();
                System.out.println("子线程结束.");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }).start();

    System.out.println("断开连接，主线程结束...");
}
}

```

```

public void connect(){
    try {
        this.cf = b.connect("127.0.0.1", 8765).sync();
        System.out.println("远程服务器已经连接，可以进行数据交换...");
    } catch (Exception e) {
        e.printStackTrace();
    }
}

//获取链接的方法
public ChannelFuture getChannelFuture(){
    //如果连接不存在或者当前连接不是活动中的重新获取链接
    if(this.cf == null){
        this.connect();
    }
    if(!this.cf.channel().isActive()){
        this.connect();
    }

    return this.cf;
}

```

6.2 心跳检测

使用 Socket 通信一般经常会处理多个服务器之间的心跳检测，肯定会有一台或几台服务

器主机，N 台从服务器（Slave），主机时刻要知道下面服务器的各个方面的情况，然后进行实时监控，这就是所谓的心跳检测。

这这里借用 Sigar（System Information Gatherer And Reporter，中文名是系统信息收集和报表工具）来读取计算机中的参数。使用前需要将对应系统的 dll 文件放入 java/bin 目录下，具体的 API 在 test 包下。

核心代码如下：

1. ClientHeartBeattHandler

```
public class ClientHeartBeatHandler extends SimpleChannelInboundHandler<String> {
    //创建可执行定时任务的线程池，初始化核心线程数为1
    private ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(1);
    //Future模式，异步获取资源，表示ScheduledExecutorService返回的结果
    private ScheduledFuture<?> heartBeat;
    //主动向服务器发送认证信息
    private InetAddress addr ;
    //用于模拟对Server返回信息的验证
    private static final String SUCCESS_KEY = "auth_success_key";

    @Override
    public void channelActive(ChannelHandlerContext ctx) throws Exception {
        addr = InetAddress.getLocalHost();
        String ip = addr.getHostAddress();
        String key = "12345";
        //模拟证书，进行安全认证
        String auth = ip + "," + key;
        ctx.writeAndFlush(auth);
    }

    @Override
    public void channelRead0(ChannelHandlerContext ctx, String msg) throws Exception {
        if(msg instanceof String){
            String ret = (String)msg;
            if(SUCCESS_KEY.equals(ret)){
                //握手成功，主动发送心跳消息 参数意义：从第0秒开始每两秒执行 HeartBeatTask的任务
                this.heartBeat = this.scheduler.scheduleWithFixedDelay(new HeartBeatTask(ctx), 0, 2, TimeUnit.SECONDS);
                System.out.println(msg);
            }
            else {
                System.out.println(msg);
            }
        }
    }
}
```

```

public HeartBeatTask(final ChannelHandlerContext ctx) {
    this.ctx = ctx;
}

@Override
public void run() {
    try {
        //自定义的请求对象，用于封装计算机运行数据
        ReqInfo info = new ReqInfo();
        //ip
        info.setIp(addr.getHostAddress());
        Sigar sigar = new Sigar();
        //cpu
        CpuPerc cpuPerc = sigar.getCpuPerc();
        HashMap<String, Object> cpuPercMap = new HashMap<String, Object>();
        cpuPercMap.put("combined", cpuPerc.getCombined());
        cpuPercMap.put("user", cpuPerc.getUser());
        cpuPercMap.put("sys", cpuPerc.getSys());
        cpuPercMap.put("wait", cpuPerc.getWait());
        cpuPercMap.put("idle", cpuPerc.getIdle());
        // 内存
        Mem mem = sigar.getMem();
        HashMap<String, Object> memoryMap = new HashMap<String, Object>();
        memoryMap.put("total", mem.getTotal() / 1024L);
        memoryMap.put("used", mem.getUsed() / 1024L);
        memoryMap.put("free", mem.getFree() / 1024L);
        info.setCpuPercMap(cpuPercMap);
        info.setMemoryMap(memoryMap);
        ctx.writeAndFlush(info);

    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

2. ServerHeartBeatHandler

```

public class ServerHeartBeatHandler extends ChannelInboundHandlerAdapter {

    private static HashMap<String, String> AUTH_IP_MAP = new HashMap<String, String>();
    private static final String SUCCESS_KEY = "auth_success_key";

    static {
        //初始化链接信息
        AUTH_IP_MAP.put("192.168.0.10", "12345");
    }

    private boolean auth(ChannelHandlerContext ctx, Object msg){
        //获取发送过来的证书进行校验
        String [] ret = ((String) msg).split(",");
        String auth = AUTH_IP_MAP.get(ret[0]);
        if(auth != null && auth.equals(ret[1])){
            ctx.writeAndFlush(SUCCESS_KEY);
            return true;
        } else {
            ctx.writeAndFlush("认证失败!").addListener(ChannelFutureListener.CLOSE);
            return false;
        }
    }
}

```

7 Netty 简单源码及流程分析

7.1 ByteBuf

在了解 Netty 中的 ByteBuf 之前，先了解一下 javaNIO 包下的 ByteBuffer，ByteBuffer 有以下几个不足之处

1. ByteBuffer 长度固定，一旦分配完成，它的容量不能动态扩展和收缩，当需要编码

的 POJO 对象大于 ByteBuffer 的容量时，会发生索引越界异常；

2. ByteBuffer 只有一个标识位置的指针 Position，读写的时候需要手工调用 flip() 和 rewind() 等，使用起来较为繁琐，容错率低很容易因为细节忽略而导致程序失败。
3. API 功能有限。

7.1.1 动态扩容

既然有这么多缺点，接下来对 Netty 的 ByteBuf 进行简单说明，看一下 Bytebuf 是如何实现动态扩容。

```
@Override
public ByteBuf writeByte(int value) {
    ensureAccessible();
    ensureWritable0(1);
    _setByte(writerIndex++, value);
    return this;
}
```

```
private void ensureWritable0(int minWritableBytes) {
    if (minWritableBytes <= writableBytes()) {
        return;
    }

    if (minWritableBytes > maxCapacity - writerIndex) {
        throw new IndexOutOfBoundsException(String.format(
            "writerIndex(%d) + minWritableBytes(%d) exceeds maxCapacity(%d): %s",
            writerIndex, minWritableBytes, maxCapacity, this));
    }

    // Normalize the current capacity to the power of 2.
    int newCapacity = calculateNewCapacity(writerIndex + minWritableBytes);

    // Adjust to the new capacity.
    capacity(newCapacity);
}
```

这里与集合类扩容原理相似，当进行 write 操作时，会对需要的 write 字节进行校验，如果可写的字节数小于需要写入的字节数，并且需要写入字节数小于可写的最大字节数时就对缓冲区进行动态扩容。

再看计算容量方法也就是 `calculateNewCapacity(writerIndex + minWritableBytes)` 方法

```
private int calculateNewCapacity(int minNewCapacity) {
    final int maxCapacity = this.maxCapacity;
    final int threshold = 1048576 * 4; // 4 MiB page

    if (minNewCapacity == threshold) {
        return threshold;
    }

    // If over threshold, do not double but just increase by threshold
    if (minNewCapacity > threshold) {
        int newCapacity = minNewCapacity / threshold * threshold;
        if (newCapacity > maxCapacity - threshold) {
            newCapacity = maxCapacity;
        } else {
            newCapacity += threshold;
        }
        return newCapacity;
    }

    // Not over threshold. Double up to 4 MiB, starting from 64.
    int newCapacity = 64;
    while (newCapacity < minNewCapacity) {
        newCapacity <<= 1;
    }

    return Math.min(newCapacity, maxCapacity);
}
```

首先判断当前传入的大小是否小于 64，否则就返回 64，如果大于 64 且小于 threshold 就每次增大 2 倍。否则就每次添加 4m 或者当新需要的空间大于最大空间减去 4m 时，就直接赋值最大的空间。然后再调用 `capacity(newCapacity)` 方法进行扩容。

下面简单了解一下 ByteBuf 的结构图：



从内存分配角度看，ByteBuf 可以分为两类：

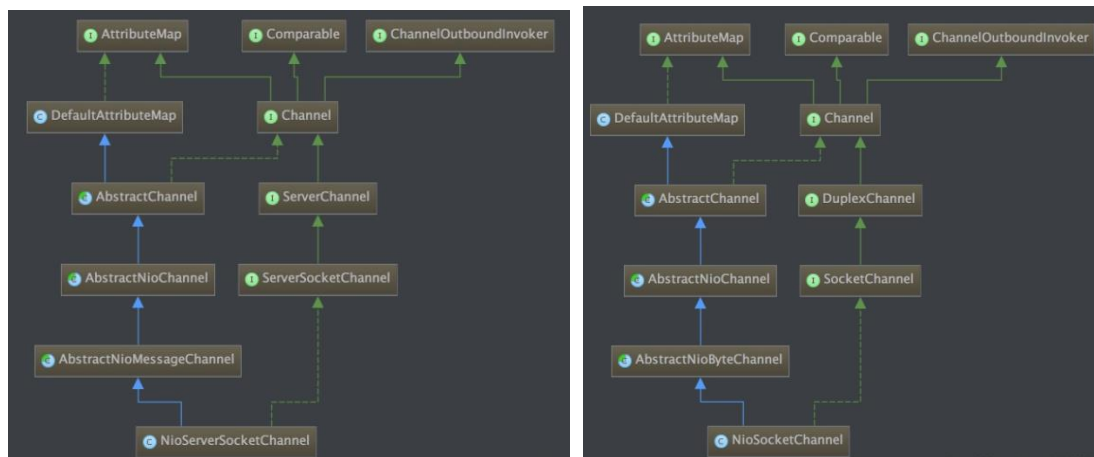
1. 堆内存字节缓冲区：特点是内存分配和回收速度快，可以被 JVM 自动回收；缺点是如果进行 Socket 的 I/O 读写，需要额外做一次内存复制，将堆内存对应的缓冲区复制到内核 Channel 中，性能有一定程度的下降。
2. 直接内存字节缓冲区：非堆内存，在堆外进行分配，相比于堆内存，分配和回收速度慢一些，但是写入或者从 Socket Channel 中读取时，由于少一次内存复制，速度比堆内存块。

在资料中有表明，ByteBuf 的最佳实践是在 I/O 通信线程的读写缓冲区使用 DirectByteBuf，后端业务消息的编码模块使用 HeapByteBuf，这样的组合可以达到性能优化。

7.2 Channel 和 Unsafe

`io.netty.channel.Channel` 是 Netty 网络操作抽象类，聚合了一组功能，包括但不限于网络读写，客户端发起链接，主动关闭链接，链路关闭，获取通信双方的网络地址等，也可以通过它获取 `Channel` 和 `EventLoop`，获取缓冲区分配器 `ByteBufAllocator` 和 `pipeline` 等。

7.2.1 Channel



Channel 主要继承类图，`NioServerSocketChannel`(左), `NioSocketChannel`(右)

7.2.1.1 AbstractChannel

`AbstractChannel` 聚合了所有 `Channel` 使用到的能力对象，由 `AbstractChannel` 提供初始化和统一封装，如果功能和子类强相关，则定义成抽象方法由子类具体实现。

Netty 基于事件驱动，可以为当 `Channel` 进行 I/O 操作时会产生对应的 I/O 事件，然后驱动事件在 `ChannelPipeline` 中传播，由对应的 `ChannelHandler` 对事件进行拦截和处理，可以通过事件定义来划分事件拦截切面，相当于 AOP，但是性能更高。

7.2.1.2 AbstractNioChannel

1. 成员变量

```

private final SelectableChannel ch; 用于设置SelectableChannel参数和进行I/O操作
protected final int readInterestOp;
volatile SelectionKey selectionKey; SelectionKey是注册到EventLoop后返回的选择键。
private volatile boolean inputShutdown;
private volatile boolean readPending;

/**
 * The future of the current connection attempt. If not null, subsequent
 * connection attempts will fail.
 */
private ChannelPromise connectPromise; 连接操作结果
private ScheduledFuture<?> connectTimeoutFuture; 连接超时定时器
private SocketAddress requestedRemoteAddress; 请求的通信地址信息

```

2. 核心 API

Channel 的注册实现。

```

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

```

定义一个布尔类型的局部变量 `selected` 来标识注册操作是否成功，调用 `SelectableChannel` 的 `register` 方法，将当前的 Channel 注册到 `EventLoop` 的多路复用器上。

注册 Channel 的时候需要制定监听的网络操作位来标识 Channel 对哪几类网络时间感兴趣（这里与 JDK 的 `SelectionKey` 的状态一致位于 `java.nio.channels.SelectionKey`）。

```
public static final int OP_READ = 1 << 0;
```

读操作位

```
public static final int OP_WRITE = 1 << 2;
```

写操作位

```
public static final int OP_CONNECT = 1 << 3;
```

客户端连接服务器操作位

```
public static final int OP_ACCEPT = 1 << 4;
```

服务端接收客户端连接操作位

从下图中可以看出 `AbstractNioChannel` 注册的是 0，仅仅是完成注册操作，注册的时候可以指定附件，后续 Channel 接收到网络时间通知时可以从 `SelectionKey` 中重新获取之前的附件进行处理，此处将其实现子类作为附件注册，如果注册成功，返回 `selectionKey`，通过 `selectionKey` 可以从多路复用器中获取 Channel 对象。

```
.register(eventLoop().unwrappedSelector(), 0, this);
```

Parameters:

sel The selector with which this channel is to be registered

ops The interest set for the resulting key

att The attachment for the resulting key; may be null

7.2.1.3 NioServerSocketChannel

```
private static final ChannelMetadata METADATA = new ChannelMetadata(false);
private static final SelectorProvider DEFAULT_SELECTOR_PROVIDER = SelectorProvider.provider();

private static final InternalLogger logger = InternalLoggerFactory.getInstance(NioServerSocketChannel.class);

private static ServerSocketChannel newSocket(SelectorProvider provider) {
    try {
        /**
         * Use the {@link SelectorProvider} to open {@link SocketChannel} and so remove condition in
         * {@link SelectorProvider#provider()} which is called by each ServerSocketChannel.open() otherwise.
         * See <a href="https://github.com/netty/netty/issues/2308">#2308</a>.
         */
        return provider.openServerSocketChannel();
    } catch (IOException e) {
        throw new ChannelException(
            "Failed to open a server socket.", e);
    }
}

private final ServerSocketChannelConfig config;
```

首先创建了静态的 ChannelMetadata 成员变量，然后定义了 ServerSocketChannelConfig 用于配置 ServerSocketChannel 的 TCP 参数。静态的 newSocket 方法用于通过 ServerSocketChannel 的 open 打开新的 ServerSocketChannel 通道。

通过 java.net.ServerSocket 的 isBound 方法判断服务器端监听端口是否处于绑定状态，它的 remoteAddress 为空。服务端在进行绑定端口的时候。可以指定 backlog(Socket 参数，服务端接受连接的队列长度，如果队列已满，客户端连接将被拒绝。默认值，Windows 为 200，其他为 128。)。

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

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

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

    return 0;
}
```

通过 ServerSocketChannel 的 accept 接收新的客户端连接，如果 SocketChannel 不为空，则利用当前的 NioServerSocketChannel、EventLoop 和 SocketChannel 创建新的 NioSocketChannel，并将其加入到 List<Object>buf 中，最后返回 1，标识服务端消息读取成功。对于 NioServerSocketChannel, 它的读取操作就是接收客户端的连接，创建 NioSocketChannel 对象。

7.2.1.4 NioSocketChannel

```
@Override
protected boolean doConnect(SocketAddress remoteAddress, SocketAddress localAddress) throws Exception {
    if (localAddress != null) {
        doBind0(localAddress);
    }

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

这里了解客户端连接相关 API 实现，判断本地的 Socket 地址是否为空，如果不为空则调用 `java.nio.channels.SocketChannel.socket().bind()` 方法绑定本地地址。如果绑定成，则继续调用 `java.nio.channels.SocketChannel.connect(SocketAddress remote)` 发起 TCP 链接。

对于读写的操作，这里进行简单的描述，就是从 `SocketChannel` 中读取 L 个字节长度到 `ByteBuffer` 中，L 为 `ByteBuffer` 可写的字节数，从 `SocketChannel` 中读取字节数到缓冲区。再传递到 Server 端。

7.3.1 Unsafe

`Unsafe` 接口是 `Channel` 接口的辅助接口，I/O 的读写操作都是由 `Unsafe` 接口负责完成的。

7.3.1.1 AbstractUnsafe

1. Register 方法

```

@Override
public final void register(EventLoop eventLoop, final ChannelPromise promise) {
    if (eventLoop == null) {
        throw new NullPointerException("eventLoop");
    }
    if (isRegistered()) {
        promise.setFailure(new IllegalStateException("registered to an event loop already"));
        return;
    }
    if (!isCompatible(eventLoop)) {
        promise.setFailure(
            new IllegalStateException("incompatible event loop type: " + eventLoop.getClass().getName()));
        return;
    }

    AbstractChannel.this.eventLoop = eventLoop;

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

```

Register 方法主要用于将当前 Unsafe 对应的 Channel 注册到 EventLoop 的多路复用器上，首先判断当前所在的线程是否是 Channel 对应的 NioEventLoop 线程，如果是同一个线程，则不存在多线程并发操作问题，直接调用 register0 进行注册；如果是由用户线程或者其它线程发起的注册操作，则将注册操作封装成 Runnable，放到 NioEventLoop 任务的执行队列中执行。

Register0 方法中首先调用 Open 方法判断 Channel 是否打开，如果没有则无法注册，直接返回。校验通过后调用 doRegister 方法（AbstractNioChannel 实现）。如注册成功，判断当前 Channel 是否已经被集火，如果被集火，则调用 ChannelPipeline 的 fireChannelActive 方法。

2. bind 方法

主要用于绑定端口。

3. write 方法

将消息添加到环形发送数组中，如果链路正常，将需要发送的 msg 和 promise 放入发送缓冲区中。

7.3.1.2 NioByteUnsafe

主要简述 read 方法：

```

@Override
public final void read() {
    final ChannelConfig config = config();
    if (!config.isAutoRead() && !isReadPending()) {
        // ChannelConfig.setAutoRead(false) was called in the meantime
        removeReadOp();
        return;
    }

    final ChannelPipeline pipeline = pipeline();
    final ByteBufAllocator allocator = config.getAllocator();
    final int maxMessagesPerRead = config.getMaxMessagesPerRead();
    RecvByteBufAllocator.Handle allocHandle = this.allocHandle;
    if (allocHandle == null) {
        this.allocHandle = allocHandle = config.getRecvByteBufAllocator().newHandle();
    }
}

```

首先获取 NioSocketChannel 的 SocketChannelConfig, 主要用于客户端连接的 TCP 参数, 如果首次调用 allocHandle 将从 RecvByteBufAllocator 创建 Handle, 它有两种实现, 这里简单分析 AdaptiveRecvByteBufAllocator 实现。

当 NioSocketChannel 执行完读操作后, 会计算获得本次轮询读取的总字节数, 就是参数 actualReadBytes, 执行 record 方法, 根据实际读取字节数对 ByteBuf 进行动态扩张。

```

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

```

首先, 对当前索引做缩减, 然后获取收缩后索引对应的容量, 与实际读取的字节数进行对比, 如果发现小于收缩后的容量, 重新对索引复制, 取收缩后的和最小的中较大的作为最新索引, 然后为下一次缓冲区容量分配复制, 如果字节数大于预先分配的容量, 则进行动态扩张, 重新计算索引。

再继续分析读操作, 首先通过接受缓冲区分配的 Handler 计算获得下次预分配的缓冲区容量 byteBufCapacity, 接着根据容量进行分配, 接受 ByteBuf 分配完成后, 进行消息的异步读取, **int localReadAmount = doReadBytes(byteBuf);** 返回本次可读的最大长度, 跟进方法

```

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

```

再跟进 writeBytes 方法

```

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

```

继续查看 setBytes 方法

```

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

```

SocketChannel 的 read 方法参数是 javaNIO 的 ByteBuffer, 所以讲 Netty 的 ByteBuf 转换为 JDK 的 ByteBuffer, 随后调用 ByteBuffer 的 clear 方法对指针进行重置用于新消息的读取, 返回读取的字节数, 如果读取的字节数小于或者等于 0, 表示没有就绪消息可读或者发生了 I/O 异常, 此时释放接受缓冲区; 如果小于 0 需要将 close 状态调整为 true, 用于关闭连接, 释放句柄资源, 完成后退出循环。

```

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

```

触发和完成 ChannelRead 时间调用之后, 将接受缓冲区释放。

```

pipeline.fireChannelRead(byteBuf);
byteBuf = null;

```

读操作在循环体中进行, 每次读取操作完成之后, 会对读取的字节数进行累加

```

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

totalReadAmount += localReadAmount;

```

累加之前, 对长度上限做保护, 如果累计读取的字节数已经发生移出, 则将读取到的字节数设置为整型的最大长度, 然后退出循环, 如果没有继续累加, 最后再进行判断是否小于缓冲区可写的容量, 是则退出循环, 否则继续执行读操作, 当超过连续读操作上限时()需要强制退出, 等待下一次 selector 轮询。

完成读操作后, 触发 ChannelReadComplete 事件, 调用接收缓冲区容量分配器的 Hanlder 的记录方法, 将读取到的字节数传入 record 方法中进行缓冲区的动态分配。

7.3 ChannelPipeline 和 ChannelHandler

ChannelPipeline 和 ChannelHandler 机制类似于 Servlet 和 Filter 过滤器，这类拦截器实际上是责任链模式的变形，主要为了方便事件的拦截和用户业务逻辑的定制。Netty 将 Channel 的数据管道抽象为 ChannelPipeline，

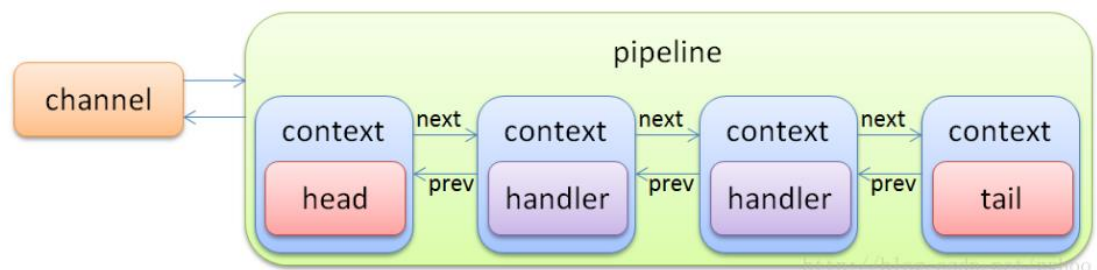
```
public interface ChannelPipeline
    extends Iterable<Entry<String, ChannelHandler>> {
```

消息在 ChannelPipeline 中流动和传递，其持有 I/O 事件拦截器 ChannelHandler 的链表，由 ChannelHandler 对 I/O 事件进行拦截和处理，可以方便的通过增加删除 ChannelHandler 来实现不同的业务逻辑指定，不需要对已有的进行修改。

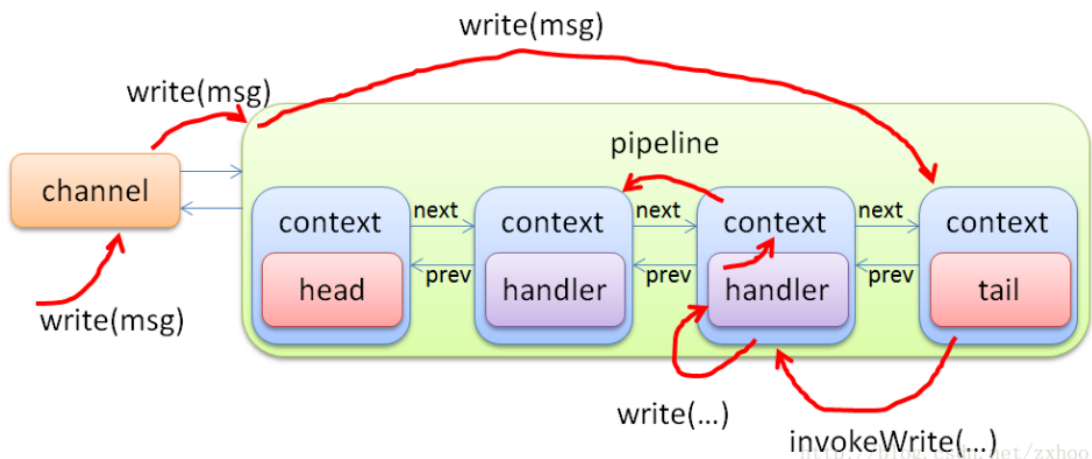
7.3.1 ChannelPipeline

消息读取和发送的全流程：

1. SocketChannel read() 方法读取 ByteBuf，出发 ChannelRead 事件，由 I/O 线程的 NioEventLoop 调用 ChannelPipeline 的 fireChannelRead(Object msg)方法，将消息传输到 ChannelPipeline 中
2. 消息被 HeadHandler、ChannelHandler 链拦截和处理，过程中任何 ChannelHandler 都可以中断当前流程，结束消息传递。



3. 调用 ChannelHandlerContext 的 write 方法发送消息，消息从 TailHandler 开始，途径 ChannelHandler 链最终被添加到消息发送缓冲区中等待刷新和发送，在此过程中也可以中断消息的传递。构造异常的 Future 返回。



7.3.2 ChannelHandler

ChannelHandler 支持注解，目前支持注解由两种。

- Sharable: 多个 ChannelPipeline 公用一个 ChannelHandler;
- Skip: 被 Skip 注解的方法不会被调用。

ChannelHandler 是 Netty 框架和用户代码主要定制点，所以种类比较多，系统的 ChannelHandler 主要分类：

- ChannelPipeline 的系统 ChannelHandler，用于 I/O 操作和对事件预处理主要包括 HeadHandler 和 TailHandler;
- 编解码，主要包括 ByteToMessageCodec、MessageToMessageDecoder 等
- 其他系统性，包括流量整形、读写超时、日志等。

这里提供参考资料(详见参考资料[14])。不再进一步描述。

7.4 EventLoop 和 EventLoopGroup

这里主要关注 Netty 的线程模型，和 NioEventLoop 相关源码。

7.4.1 Netty 线程模型

Netty 线程模型并不是一成不变的，主要还是取决于用户的启动参数配置，通过设置不同的启动参数，Netty 可以支持 Reactor 单线程模型、多线程模型和主从 Reactor 多线程模型（Reactor 模型将在附录中进行描述）。

可以通过 HelloWorld 程序 Server 端来分析其线程模型

```

public class Server {
    public static void main(String[] args) throws Exception {
        // 第一个线程组, 用于接受Client端连接
        EventLoopGroup bossGroup = new NioEventLoopGroup();
        // 第二个线程组 用于实际业务的处理操作
        EventLoopGroup workerGroup = new NioEventLoopGroup();

        try {
            // 辅助类, 用于配置Server
            ServerBootstrap b = new ServerBootstrap();
            // 将工作线程组加入进来
            b.group(bossGroup, workerGroup)
            // 指定NioServerSocketChannel类型的通道
            .channel(NioServerSocketChannel.class)
            // 配置日志输出
            .handler(new LoggingHandler(LogLevel.INFO))
            // 绑定具体的事件处理器
            .childHandler(new ChannelInitializer<SocketChannel>() {
                @Override
                protected void initChannel(SocketChannel ch) throws Exception {
                    ch.pipeline().addLast(new ServerHandler());
                }
            })
            .option(ChannelOption.SO_BACKLOG, 128)
            .childOption(ChannelOption.SO_KEEPALIVE, true);
            // 绑定端口, 等待绑定成功
            ChannelFuture f = b.bind(12345).sync();
            // 等待服务器退出
            f.channel().closeFuture().sync();
        } finally {
            workerGroup.shutdownGracefully();
            bossGroup.shutdownGracefully();
        }
    }
}

```

在服务端启动的时候, 创建了两个 `NioEventLoopGroup`, 他们是两个独立的线程池。一个用于接收客户端的 TCP 连接, 另一个用于处理 I/O 相关的读写操作, 或者执行系统 Task 任务, 定时任务 Task 等。

用于接收客户端请求的线程池职责为 a) 接收客户端 TCP 连接, 初始化 Channel 参数; b) 将链路状态变更时间通知给 `ChannelPipeline`。

处理 I/O 操作的 `Reactor` 线程池职责: a) 异步读取通信端的数据包, 发送读事件到 `ChannelPipeline`; b) 异步发送消息到通信对端, 调用 `ChannelPipeline` 的消息发送接口; c) 执行系统调用 Task; d) 执行定时任务 Task, 例如链路空闲状态监测定时任务。

通过调整线程池的线程个数、是否共享线程池等方式, `Netty` 的 `Reactor` 线程模型可自由切换。

在 `NioEventLoop` 读取到消息后, 直接调用 `ChannelPipeline` 的 `fireChannelRead(Object msg)`。只要用户不主动切换线程, 一直都是由 `NioEventLoop` 调用用户的 `Handler`, 期间不进行线程切换。避免了锁竞争。

7.4.2 NioEventLoop

在上面线程模型中已经描述了 `NioEventLoop` 的作用。这里做简要源码分析

作为 NIO 框架的 `Reactor` 线程, `NioEventLoop` 需要处理网络 I/O 事件, 因此必须聚合一个多路复用器对象也就是 `Selector`。


```
private Selector selector;
private Selector unwrappedSelector;
private SelectedSelectionKeySet selectedKeys;

private final SelectorProvider provider;
```

在构造方法中对其初始化，调用 `Selector.open()` 方法创建并打开一个新的 `Selector`，它对 `selectedKeys` 进行了优化，可以通过 `io.netty.noKeySetOptimization` 开关决定是否启用，默认是不打开的。

```
private Selector openSelector() {
    try {
        unwrappedSelector = provider.openSelector();
    } catch (IOException e) {
        throw new ChannelException("failed to open a new selector", e);
    }

    if (DISABLE_KEYSET_OPTIMIZATION) {
        return unwrappedSelector;
    }
}
```

如果没有开启优化开关，通过 `provider` 的 `openSelector` 创建并打开多路复用器后直接返回没有包装的 `selector` 对象，如果开启了优化开关，将通过反射的方式从 `Channel` 获取其对象，从而返回包装后的 `SelectedKeySet` 将原 JDK 的 `selectedKey` 替换。

再看 `run` 方法的实现，只有当 `NioEventLoop` 接收到退出指令的时候才退出循环，否则一直执行下去。

```
for (;;) {
    try {
        switch (selectStrategy.calculateStrategy(selectNowSupplier, hasTasks())) {
            case SelectStrategy.CONTINUE:
                continue;
            case SelectStrategy.SELECT:
                select();
                break;
        }
    } catch (Exception e) {
        handleException(e);
    }
}
```

首先通过 `hasTasks` 方法判断当前消息队列中是否有消息需要处理，有则立即出发 `Selector` 选择操作，有准备就绪的 `Channel` 就返回就绪的 `Channel` 的状态，否则就返回-1，完成之后再判断是否调用 `wakeup` 方法，如果调用，执行 `selector.wakeup()`。如果没有消息处理，由多路复用器轮询，看是否有准备就绪的 `Channel`，如果轮询到处于就绪状态的 `SocketChannel` 则需要处理网络 I/O 事件，对 `SocketChannel` 的附件类型进行判读，如果是 `AbstractNioChannel` 类型说明是 `NioServerSocketChannel` 或者 `NioSocketChannel`，需要进行 I/O 读写操作，获取内部类 `Unsafe`，如果是读或者链接操作，调用 `read` 方法，如果是 `NioServerSocketChannel` 读操作就是接收客户端的 TCP 连接，对于 `NioSocketChannel`，它的读操作就是 `SocketChannel` 中读取的 `ByteBuffer` 如果是写调用 `flush` 进行发送，如果连接，需要对连接进行判读，最后判断是或否进入优雅停机状态。便利 `Channel` 调用 `Unsafe.close()` 方法关闭链路，释放线程池等资源。

8 参考资料

- [1]. 李林锋.Netty权威指南 第二版[M].2014.06.01
- [2]. Reactor模式详解 <http://www.blogjava.net/DLevin/archive/2015/09/02/427045.html>
- [3]. Netty系列之Netty高性能 <http://www.infoq.com/cn/articles/netty-high-performance/>
- [4]. 开发第一个Netty应用程序 <http://www.2cto.com/kf/201405/299028.html>
- [5]. Norman Maurer, Marvin Allen Wolfthal ,译者: 桃小胖. Netty In Action 中文版[M]. 2015
- [6]. Netty线程模型 <http://dwz.cn/5KE9LW>
- [7]. Netty 用户指南 <http://ifeve.com/netty5-user-guide/>
- [8]. Netty学习之旅----ByteBuf内部结构与API学习 <http://blog.csdn.net/prestigeding/article/details/53980790>
- [9]. 设计模式（九）外观模式Facade（结构型）<http://blog.csdn.net/hguisu/article/details/7533759>
- [10]. 自顶向下深入分析Netty（六）--Channel总述 <http://www.jianshu.com/p/fffc18d33159>
- [11]. [netty核心类]--Channel和Unsafe类 <http://blog.csdn.net/u010853261/article/details/55565251>
- [12]. Netty4 学习笔记（1）-- ChannelPipeline <http://blog.csdn.net/zxhoo/article/details/17264263>
- [13]. ChannelHandler功能介绍 <http://www.cnblogs.com/wade-luffy/p/6222960.html>
- [14]. Java aio(异步网络IO)初探 <http://www.iteye.com/topic/472333>

9 附件

9.1 代码附件



NettyDemo.rar

目录结构:

- ▼ NettyDemo
 - ▼ src
 - ▼ aimei.netty
 - > delimiterbasedframe 分隔符方式解决粘包/拆包问题
 - > fixedlengthframe 消息定长解决粘包拆包问题
 - > heartBeat 心跳检测
 - > helloworld Netty HellowWorld程序
 - > runtime 使用ReadTimeoutHandler设置超时时间，并重新获取连接
 - > serializable Marshalling 序列化
 - > test Sigar 使用示例
 - > JRE System Library [JavaSE-1.7]
 - ▼ Referenced Libraries
 - > commons-logging-1.1.1.jar
 - > jboss-marshalling-1.3.0.CR9.jar
 - > jboss-marshalling-serial-1.3.0.CR9.jar
 - > log4j.jar
 - > sigar.jar
 - > netty-all-4.0.45.Final.jar
 - > netty-codec-4.0.45.Final.jar
 - > lib
 - > receive
 - > sources

10 附录(相关设计模式暂未完善)

10.1 涉及到的设计模式

10.1.1 Reactor 模式

10.1.2 Proactor 模式

10.1.3 Facade 模式

10.1.4 Feture 模式

10.2 其他

10.2.1 Channel 配置参数

通用参数	CONNECT_TIMEOUT_MILLIS	Netty 参数，连接超时毫秒数，默认值 30000 毫秒即 30 秒。
	MAX_MESSAGES_PER_READ	Netty 参数，一次 Loop 读取的最大消息数，对于 ServerChannel 或者 NioByteChannel，默认值为 16，其他 Channel 默认值为 1。默认值这样设置，是因为：ServerChannel 需要接受足够多的连接，保证大吞吐量，NioByteChannel 可以减少不必要的系统调用 select。
	WRITE_SPIN_COUNT	Netty 参数，一个 Loop 写操作执行的最大次数，默认值为 16。也就是说，对于大数据量的写操作至多进行 16 次，如果 16 次仍没有全部写完数据，此时会提交一个新的写任务给 EventLoop，任务将在下次调度继续执行。这样，其他的写请求才能被响应不会因为单个大数据量写请求而耽误。
	ALLOCATOR	Netty 参数，ByteBuf 的分配器，默认值为 ByteBufAllocator.DEFAULT，4.0 版本为 UnpooledByteBufAllocator，4.1 版本为 PooledByteBufAllocator。该值也可以使用系统参数 io.netty.allocator.type 配置，使用字符串值："unpooled"，"pooled"。
	RCVBUF_ALLOCATOR	Netty 参数，用于 Channel 分配接受 Buffer 的分配器，默认值为 AdaptiveRecvByteBufAllocator.DEFAULT，是一个自适应的接受缓冲区分配器，能根据接受到的数据自动调节大

		小。可选值为 FixedRecvByteBufAllocator，固定大小的接受缓冲区分配器。
	AUTO_READ	Netty 参数，自动读取，默认值为 True。Netty 只在必要的时候才设置关心相应的 I/O 事件。对于读操作，需要调用 channel.read()设置关心的 I/O 事件为 OP_READ，这样若有数据到达才能读取以供用户处理。该值为 True 时，每次读操作完毕后会自动调用 channel.read()，从而有数据到达便能读取；否则，需要用户手动调用 channel.read()。需要注意的是：当调用 config.setAutoRead(boolean)方法时，如果状态由 false 变为 true，将会调用 channel.read()方法读取数据；由 true 变为 false，将调用 config.autoReadCleared()方法终止数据读取。
	WRITE_BUFFER_HIGH_WATER_MARK	Netty 参数，写高水位标记，默认值 64KB。如果 Netty 的写缓冲区中的字节超过该值，Channel 的 isWritable()返回 False
	WRITE_BUFFER_LOW_WATER_MARK	Netty 参数，写低水位标记，默认值 32KB。当 Netty 的写缓冲区中的字节超过高水位之后若下降到低水位，则 Channel 的 isWritable()返回 True。写高低水位标记使用户可以控制写入数据速度，从而实现流量控制。推荐做法是：每次调用 channel.write(msg)方法首先调用 channel.isWritable()判断是否可写。
	MESSAGE_SIZE_ESTIMATOR	Netty 参数，消息大小估算器，默认为 DefaultMessageSizeEstimator.DEFAULT。估算 ByteBuf、ByteBufHolder 和 FileRegion 的大小，其中 ByteBuf 和 ByteBufHolder 为实际大小，FileRegion 估算值为 0。该值估算的字节数在计算水位时使用，FileRegion 为 0 可知 FileRegion 不影响高低水位。
	SINGLE_EVENTEXECUTOR_PER_GROUP	Netty 参数，单线程执行 ChannelPipeline 中的事件，默认值为 True。该值控制执行 ChannelPipeline 中执行 ChannelHandler 的线程。如果为 True，整个 pipeline 由一个线程执行，这样不需要进行线程切换以及线程同步，是 Netty4 的推荐做法；如果为 False，ChannelHandler 中的处理过程会由 Group 中的不同线程执行。
SocketChannel 参数	SO_RCVBUF	Socket 参数，TCP 数据接收缓冲区大小。该缓冲区即 TCP 接收滑动窗口，linux 操作系统可使用命令：cat /proc/sys/net/ipv4/tcp_rmem 查询其大小。一般情况下，该值可由用户在任意时刻设置，但当设置值超过 64KB 时，需要在连接到远端之前设置。
	SO_SNDBUF	Socket 参数，TCP 数据发送缓冲区大小。该缓冲区即 TCP 发送滑动窗口，linux 操作系统可使用命令：cat /proc/sys/net/ipv4/tcp_wmem 查询其大小。
	TCP_NODELAY	TCP 参数，立即发送数据，默认值为 True（Netty 默认为 True 而操作系统默认为 False）。该值设置 Nagle 算法的启用，改算法将小的碎片数据连接成更大的报文来最小化所发送的报文的数量，如果需要发送一些较小的报文，则需要禁用

		该算法。Netty 默认禁用该算法，从而最小化报文传输延时。
	SO_KEEPALIVE	Socket 参数，连接保活，默认值为 False。启用该功能时，TCP 会主动探测空闲连接的有效性。可以将此功能视为 TCP 的心跳机制，需要注意的是：默认的心跳间隔是 7200s 即 2 小时。Netty 默认关闭该功能。
	SO_REUSEADDR	Socket 参数，地址复用，默认值 False。有四种情况可以使用：(1).当有一个有相同本地地址和端口的 socket1 处于 TIME_WAIT 状态时，而你希望启动的程序的 socket2 要占用该地址和端口，比如重启服务且保持先前端口。(2).有多块网卡或用 IP Alias 技术的机器在同一端口启动多个进程，但每个进程绑定的本地 IP 地址不能相同。(3).单个进程绑定相同的端口到多个 socket 上，但每个 socket 绑定的 ip 地址不同。(4).完全相同的地址和端口的重复绑定。但这只用于 UDP 的多播，不用于 TCP。
	SO_LINGER	Socket 参数，关闭 Socket 的延迟时间，默认值为-1，表示禁用该功能。-1 表示 socket.close()方法立即返回，但 OS 底层会将发送缓冲区全部发送到对端。0 表示 socket.close()方法立即返回，OS 放弃发送缓冲区的数据直接向对端发送 RST 包，对端收到复位错误。非 0 整数值表示调用 socket.close()方法的线程被阻塞直到延迟时间到或发送缓冲区中的数据发送完毕，若超时，则对端会收到复位错误。
	IP_TOS	IP 参数，设置 IP 头部的 Type-of-Service 字段，用于描述 IP 包的优先级和 QoS 选项。
	ALLOW_HALF_CLOSURE	Netty 参数，一个连接的远端关闭时本地端是否关闭，默认值为 False。值为 False 时，连接自动关闭；为 True 时，触发 ChannelInboundHandler 的 userEventTriggered()方法，事件为 ChannelInputShutdownEvent。
ServerSocketChannel 参数	SO_RCVBUF	已说明，需要注意的是：当设置值超过 64KB 时，需要在绑定到本地端口前设置。该值设置的是由 ServerSocketChannel 使用 accept 接受的 SocketChannel 的接收缓冲区。
	SO_REUSEADDR	上文已描述
	SO_BACKLOG	Socket 参数，服务端接受连接的队列长度，如果队列已满，客户端连接将被拒绝。默认值，Windows 为 200，其他为 128。
DatagramChannel 参数	SO_BROADCAST	Socket 参数，设置广播模式。
	SO_RCVBUF	上文已描述
	SO_SNDBUF	上文已描述
	SO_REUSEADDR	上文已描述
	IP_MULTICAST	对应 IP 参数 IP_MULTICAST_LOOP，设置本地回环接口的

	ST_LOOP_DISABLED	多播功能。由于 IP_MULTICAST_LOOP 返回 True 表示关闭，所以 Netty 加上后缀_DISABLED 防止歧义。
	IP_MULTICAST_ADDR	对应 IP 参数 IP_MULTICAST_IF，设置对应地址的网卡为多播模式。
	IP_MULTICAST_IF	对应 IP 参数 IP_MULTICAST_IF2，同上但支持 IPV6。
	IP_MULTICAST_TTL	IP 参数，多播数据报的 time-to-live 即存活跳数。
	IP_TOS	上文已描述
	DATAGRAM_CHANNEL_ACTIVE_ON_REGISTRATION	Netty 参数，DatagramChannel 注册的 EventLoop 即表示已激活。