Broadview
www.broadview.com.cn

Netty 权威指南（第2版）

李林锋 / 著

Java高性能NIO通信首选框架

大数据时代构建高可用分布式系统利器

Netty: The Definitive Guide, 2nd Edtion

中国工信出版集团

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

内 容 简 介

《Netty 权威指南（第 2 版）》是异步非阻塞通信领域的经典之作，基于最新版本的 Netty 5.0 编写，是国内首本深入介绍 Netty 原理和架构的书籍，也是作者多年实战经验的总结和浓缩。内容不仅包含 Java NIO 入门知识、Netty 的基础功能开发指导、编解码框架定制等，还包括私有协议栈定制和开发、Netty 核心类库源码分析，以及 Netty 的架构剖析。

本书适合架构师、设计师、软件开发工程师、测试人员以及其他对 Java NIO 框架、Netty 感兴趣的相关人士阅读，通过本书的学习，读者不仅能够掌握 Netty 基础功能的使用和开发，更能够掌握 Netty 核心类库的原理和使用约束，从而在实际工作中更好地使用 Netty。

图书在版编目（CIP）数据

Netty 权威指南 / 李林锋著. —2 版. —北京：电子工业出版社，2015.4
ISBN 978-7-121-25801-5

Ⅰ. ①N… Ⅱ. ①李… Ⅲ. ①JAVA 语言－程序设计－指南 Ⅳ. ①TP312-62

中国版本图书馆 CIP 数据核字（2015）第 067682 号

责任编辑：董 英
印 刷：北京丰源印刷厂
装 订：三河市皇庄路通装订厂
出版发行：电子工业出版社
北京市海淀区万寿路 173 信箱 邮编 100036
开 本：787×980 1/16 印张：35.75 字数：758 千字
版 次：2014 年 6 月第 1 版
2015 年 4 月第 2 版
印 次：2015 年 4 月第 1 次印刷
定 价：89.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

第13章

服务端创建

对于想要深入学习 Netty 原理的人而言，通过阅读源码是最有效的学习方式之一。尽管 Netty 使用起来并不复杂，但是通过对源码的分析和学习，掌握一些必备的基础知识还是很有必要的。

Netty 服务端创建需要的必备知识如下：

（1）熟悉 JDK NIO 主要类库的使用，例如 ByteBuffer、Selector、ServerSocketChannel 等；

（2）熟悉 JDK 的多线程编程；

（3）了解 Reactor 模式。

本文首先对 Java NIO 服务端的创建进行简单介绍，然后对 Netty 服务端的创建进行原理讲解和源码分析，以期让更多希望了解 Netty 底层原理的读者可以快速入门。

本章主要内容包括：

- 原生 NIO 类库的复杂性
- Netty 服务端创建源码分析
- 客户端接入源码分析

13.1　原生 NIO 类库的复杂性

在开始本文之前，我先讲一件自己亲身经历的事。大约在 2011 年的时候，身边有两个业务团队同时进行新版本开发，他们都需要基于 NIO 非阻塞特性构建高性能、异步和高可靠性的底层通信框架。

当时两个项目组的设计师都咨询了我的意见，在了解了两个项目团队的 NIO 编程经验和现状之后，我建议他们都使用 Netty 构建业务通信框架。令人遗憾的是其中 1 个项目组并没有按照我的建议做，而是选择直接基于 JDK 的 NIO 类库构建自己的通信框架。在他们看来，构建业务层的 NIO 通信框架并不是件难事，即便当前他们还缺乏相关经验。

两个多月过去之后，自研 NIO 框架团队的通信框架始终无法稳定地工作，他们频繁遭遇客户端断连、句柄泄露和消息丢失等问题，项目的进度出现了严重的延迟。形成鲜明对比的是，另一个团队由于基于 Netty 研发，在通信框架上节省了大量的人力和时间，加之 Netty 自身的可靠性和稳定性非常好，他们的项目进展非常顺利。

这两个项目组的不同遭遇告诉我们：开发高质量的 NIO 程序并不是一件简单的事情，除去 NIO 类库的固有复杂性和 Bug，作为 NIO 服务端，需要能够处理网络的闪断、客户端的重连、安全认证和消息的编解码、半包处理等。如果没有足够的 NIO 编程经验积累，自研 NIO 框架往往需要半年甚至数年的时间才能最终稳定下来，这种成本即便对一个大公司而言也是个严重的挑战。

13.2　Netty 服务端创建源码分析

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

Netty 为了向使用者屏蔽 NIO 通信的底层细节，在和用户交互的边界做了封装，目的就是为了减少用户开发工作量，降低开发难度。ServerBootstrap 是 Socket 服务端的启动辅助类，用户通过 ServerBootstrap 可以方便地创建 Netty 的服务端。时序图如图 13-1 所示。

13.2.1 Netty 服务端创建时序图

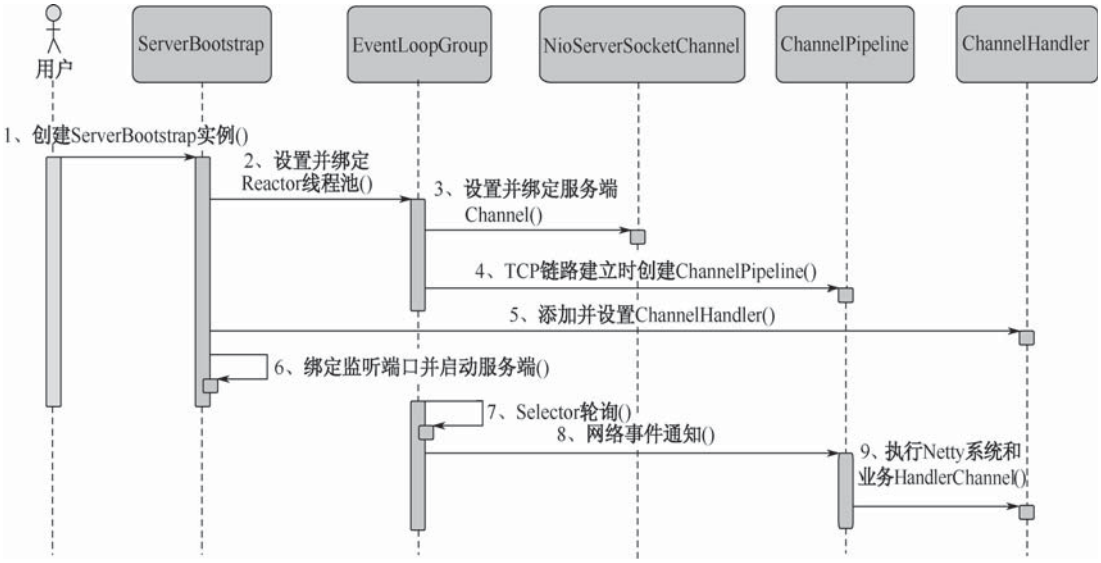

图 13-1　Netty 服务端创建时序图

下面我们对 Netty 服务端创建的关键步骤和原理进行讲解。

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

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

步骤 2：设置并绑定 Reactor 线程池。Netty 的 Reactor 线程池是 EventLoopGroup，它实际就是 EventLoop 的数组。EventLoop 的职责是处理所有注册到本线程多路复用器 Selector 上的 Channel，Selector 的轮询操作由绑定的 EventLoop 线程 run 方法驱动，在一个循环体内循环执行。值得说明的是，EventLoop 的职责不仅仅是处理网络 I/O 事件，用

户自定义的 Task 和定时任务 Task 也统一由 EventLoop 负责处理，这样线程模型就实现了统一。从调度层面看，也不存在从 EventLoop 线程中再启动其他类型的线程用于异步执行另外的任务，这样就避免了多线程并发操作和锁竞争，提升了 I/O 线程的处理和调度性能。

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

```
    public ServerBootstrap channel(Class<? extends ServerChannel> channelClass)
{
            if (channelClass == null) {
                throw new NullPointerException("channelClass");
            }
            return                                          channelFactory(new
ServerBootstrapChannelFactory<ServerChannel>(channelClass));
        }
```

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

（1）链路注册；

（2）链路激活；

（3）链路断开；

（4）接收到请求消息；

（5）请求消息接收并处理完毕；

（6）发送应答消息；

（7）链路发生异常；

（8）发生用户自定义事件。

步骤 5：初始化 ChannelPipeline 完成之后，添加并设置 ChannelHandler。ChannelHandler 是 Netty 提供给用户定制和扩展的关键接口。利用 ChannelHandler 用户可以完成大多数的功能定制，例如消息编解码、心跳、安全认证、TSL/SSL 认证、流量控制和流量整形等。Netty 同时也提供了大量的系统 ChannelHandler 供用户使用，比较实用的系统 ChannelHandler 总结如下。

（1）系统编解码框架——ByteToMessageCodec；

（2）通用基于长度的半包解码器——LengthFieldBasedFrameDecoder;

（3）码流日志打印 Handler——LoggingHandler；

（4）SSL 安全认证 Handler——SslHandler；

（5）链路空闲检测 Handler——IdleStateHandler；

（6）流量整形 Handler——ChannelTrafficShapingHandler;

（7）Base64 编解码——Base64Decoder 和 Base64Encoder。

创建和添加 ChannelHandler 的代码示例如下。

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

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

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

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

```
    private void select() throws IOException {
          Selector selector = this.selector;
          try {

                //此处代码省略...
                int selectedKeys = selector.select(timeoutMillis);
                selectCnt ++;
                //此处代码省略...
    }
```

步骤 8：当轮询到准备就绪的 Channel 之后，就由 Reactor 线程 NioEventLoop 执行 ChannelPipeline 的相应方法，最终调度并执行 ChannelHandler，接口如图 13-2 所示。

图 13-2　调度相关方法

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

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

13.2.2　Netty 服务端创建源码分析

首先通过构造函数创建 ServerBootstrap 实例，随后，通常会创建两个 EventLoopGroup

（并不是必须要创建两个不同的 EventLoopGroup，也可以只创建一个并共享），代码如下。

```
    EventLoopGroup acceptorGroup = new NioEventLoopGroup();
    EventLoopGroup IOGroup = new NioEventLoopGroup();
```

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

```
    public ServerBootstrap group(EventLoopGroup parentGroup, EventLoopGroup
childGroup) {
          super.group(parentGroup);
          if (childGroup == null) {
             throw new NullPointerException("childGroup");
          }
          if (this.childGroup != null) {
             throw new IllegalStateException("childGroup set already");
          }
          this.childGroup = childGroup;
          return this;
       }
```

其中父 NioEventLoopGroup 被传入了父类构造函数中，代码如下。

```
     public B group(EventLoopGroup group) {
          if (group == null) {
             throw new NullPointerException("group");
          }
          if (this.group != null) {
             throw new IllegalStateException("group set already");
          }
          this.group = group;
          return (B) this;
       }
```

该方法会被客户端和服务端重用，用于设置工作 I/O 线程，执行和调度网络事件的读写。

线程组和线程类型设置完成后，需要设置服务端 Channel 用于端口监听和客户端链路接入。Netty 通过 Channel 工厂类来创建不同类型的 Channel，对于服务端，需要创建 NioServerSocketChannel。所以，通过指定 Channel 类型的方式创建 Channel 工厂。

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

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

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

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

backlog 指定了内核为此套接口排队的最大连接个数，对于给定的监听套接口，内核要维护两个队列：未链接队列和已连接队列，根据 TCP 三路握手过程中三个分节来分隔这两个队列。服务器处于 listen 状态时，收到客户端 syn 分节（connect）时在未完成队列中创建一个新的条目，然后用三路握手的第二个分节即服务器的 syn 响应客户端，此条目在第三个分节到达前（客户端对服务器 syn 的 ack）一直保留在未完成连接队列中，如果三路握手完成，该条目将从未完成连接队列搬到已完成连接队列尾部。当进程调用 accept 时，从已完成队列中的头部取出一个条目给进程，当已完成队列为空时进程将睡眠，直到有条目在已完成连接队列中才唤醒。backlog 被规定为两个队列总和的最大值，大多数实现默认值为 5，但在高并发 Web 服务器中此值显然不够，Lighttpd 中此值达到 128×8。需要设置此值更大一些的原因是未完成连接队列的长度可能因为客户端 syn 的到达及等待三路握手第三个分节的到达延时而增大。Netty 默认的 backlog 为 100，当然，用户可以修改默认值，这需要根据实际场景和网络状况进行灵活设置。

TCP 参数设置完成后，用户可以为启动辅助类和其父类分别指定 Handler。两类 Handler 的用途不同：子类中的 Handler 是 NioServerSocketChannel 对应的 ChannelPipeline 的 Handler；父类中的 Handler 是客户端新接入的连接 SocketChannel 对应的 ChannelPipeline 的 Handler。两者的区别可以通过图 13-3 来展示。

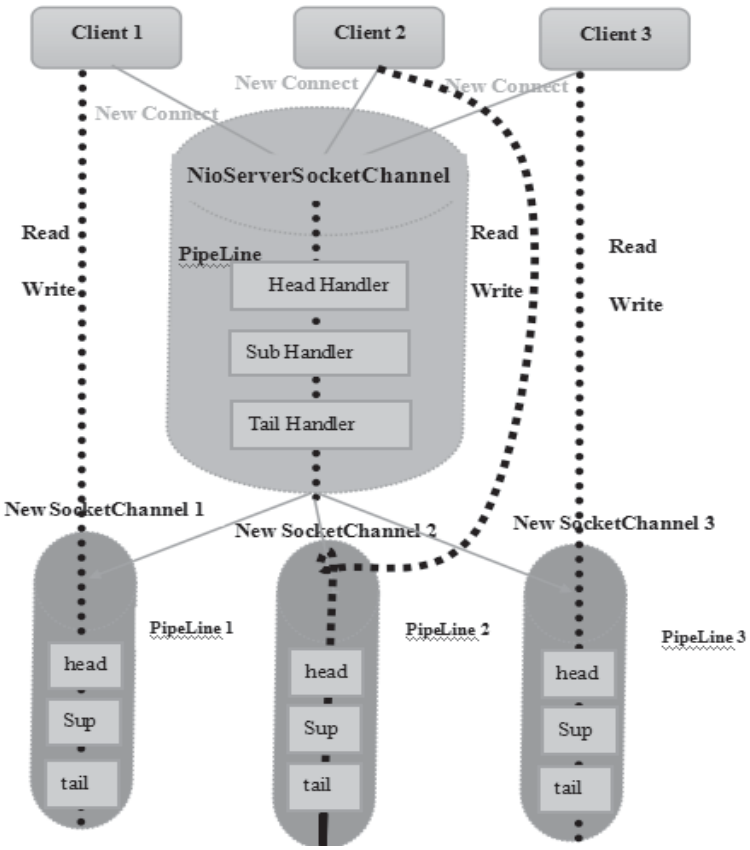

图 13-3　ServerBootstrap 的 Hanlder 模型

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

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

```
    private ChannelFuture doBind(final SocketAddress localAddress) {
           final ChannelFuture regFuture = initAndRegister();NO. 1
           final Channel channel = regFuture.channel();
           if (regFuture.cause() != null) {
               return regFuture;
           }
```

```
            final ChannelPromise promise;
            if (regFuture.isDone()) { NO. 2
                promise = channel.newPromise();
                doBind0(regFuture, channel, localAddress, promise);
            } else {
                promise       =       new       DefaultChannelPromise(channel,
GlobalEventExecutor.INSTANCE);
                regFuture.addListener(new ChannelFutureListener() {
                    @Override NO. 3
                    public void operationComplete(ChannelFuture future) throws
Exception {
                        doBind0(regFuture, channel, localAddress, promise);
                    }
                });
            }
            return promise;
        }
```

先看下 NO.1。首先创建 Channel，createChannel 由子类 ServerBootstrap 实现，创建新的 NioServerSocketChannel。它有两个参数：参数 1 是从父类的 NIO 线程池中顺序获取一个 NioEventLoop，它就是服务端用于监听和接收客户端连接的 Reactor 线程；参数 2 是所谓的 workerGroup 线程池，它就是处理 I/O 读写的 Reactor 线程组，相关代码如下。

```
     final ChannelFuture initAndRegister() {
            Channel channel;
            try {
                channel = createChannel();
            } catch (Throwable t) {
                return VoidChannel.INSTANCE.newFailedFuture(t);
            }
            try {
                init(channel);
            } catch (Throwable t) {
                channel.unsafe().closeForcibly();
                return channel.newFailedFuture(t);
            }
           //后续代码省略.....
        }
```

NioServerSocketChannel 创建成功后，对它进行初始化，初始化工作主要有以下三点。

（1）设置 Socket 参数和 NioServerSocketChannel 的附加属性，代码如下。

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

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

（2）将 AbstractBootstrap 的 Handler 添加到 NioServerSocketChannel 的 ChannelPipeline 中，代码如下。

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

（3）将用于服务端注册的 Handler ServerBootstrapAcceptor 添加到 ChannelPipeline 中，代码如下。

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

到此，Netty 服务端监听的相关资源已经初始化完毕，就剩下最后一步——注册 NioServerSocketChannel 到 Reactor 线程的多路复用器上，然后轮询客户端连接事件。在分析注册代码之前，我们先通过图 13-4 看看目前 NioServerSocketChannel 的 ChannelPipeline 的组成。

图 13-4　NioServerSocketChannel 的 ChannelPipeline

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

```
public final void register(final ChannelPromise promise) {
         if (eventLoop.inEventLoop()) {
            register0(promise);
         } else {
            try {
               eventLoop.execute(new Runnable() {
                  public void run() {
                     register0(promise);
                  }
               });
            } catch (Throwable t) {
           //此处代码省略......
      }
```

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

```
private void register0(ChannelPromise promise) {
         try {
            if (!ensureOpen(promise)) {
               return;
            }
            doRegister();
            registered = true;
            promise.setSuccess();
            pipeline.fireChannelRegistered();
            if (isActive()) {
               pipeline.fireChannelActive();
            }
         } catch (Throwable t) {
```

```
            //此处代码省略...
        }
```

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

```
    protected void doRegister() throws Exception {
        boolean selected = false;
        for (;;) {
            try {
                selectionKey = javaChannel().register(eventLoop().selector,
0, this);
    //此处代码省略...
    }
```

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

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

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

注册成功之后，触发 ChannelRegistered 事件，方法如下。

```
                promise.setSuccess();
                pipeline.fireChannelRegistered();
```

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

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

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

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

isActive()也是个多态方法。如果是服务端，判断监听是否启动；如果是客户端，判断

TCP 连接是否完成。ChannelActive 事件在 ChannelPipeline 中传递，完成之后根据配置决定是否自动触发 Channel 的读操作，代码如下。

```
    public ChannelPipeline fireChannelActive() {
          head.fireChannelActive();
          if (channel.config().isAutoRead()) {
              channel.read();
          }
          return this;
      }
```

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

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

继续看 AbstractUnsafe 的 beginRead 方法，代码如下。

```
    public void beginRead() {
             if (!isActive()) {
                 return;
             }
             try {
                 doBeginRead();
    //后续代码省略......
    }
```

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

```
    protected void doBeginRead() throws Exception {
          //以上代码省略......
          final int interestOps = selectionKey.interestOps();
          if ((interestOps & readInterestOp) == 0) {
              selectionKey.interestOps(interestOps | readInterestOp);
          }
      }
```

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

JDK SelectionKey 有 4 种操作类型，分别为：

（1）OP_READ = 1 << 0；

（2）OP_WRITE = 1 << 2；

（3）OP_CONNECT = 1 << 3；

（4）OP_ACCEPT = 1 << 4。

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

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

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

到此，服务端监听启动部分源码已经分析完成，下一章节，让我们继续分析一个新的客户端是如何接入的。

13.3 客户端接入源码分析

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

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

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

```
public void read() {
        //代码省略......
        final ChannelConfig config = config();
        final int maxMessagesPerRead = config.getMaxMessagesPerRead();
        final boolean autoRead = config.isAutoRead();
        final ChannelPipeline pipeline = pipeline();
        boolean closed = false;
        Throwable exception = null;
        try {
            for (;;) {
                int localRead = doReadMessages(readBuf);
                if (localRead == 0) {
                    break;
                }
                if (localRead < 0) {
                    closed = true;
                    break;
                }
                if (readBuf.size() >= maxMessagesPerRead | !autoRead) {
                    break;
                }
            //此处代码省略......
    }
```

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

```
    protected int doReadMessages(List<Object> buf) throws Exception {
           SocketChannel ch = javaChannel().accept();
           try {
               if (ch != null) {
                   buf.add(new                              NioSocketChannel(this,
childEventLoopGroup().next(), ch));
                   return 1;
               }
           } catch (Throwable t) {
           //后续代码省略...
       }
```

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

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

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

```
     public void channelRead(ChannelHandlerContext ctx, Object msg) {
              Channel child = (Channel) msg;
              child.pipeline().addLast(childHandler);
              //代码省略...
              child.unsafe().register(child.newPromise());
           }
```

该方法主要分为如下三个步骤。

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

第二步：设置客户端 SocketChannel 的 TCP 参数；

第三步：注册 SocketChannel 到多路复用器。

以上三个步骤执行完成之后，下面我们展开看下 NioSocketChannel 的 register 方法，代码如图 13-5 所示。

图 13-5　NioSocketChannel 的 register 方法实现

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

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

HeadHandler read()方法的代码已经在之前的小节介绍过，用来将网络操作位修改为读操作。创建 NioSocketChannel 的时候已经将 AbstractNioChannel 的 readInterestOp 设置为 OP_READ，这样，执行 selectionKey.interestOps(interestOps | readInterestOp)操作时就会把操作位设置为 OP_READ。代码如下。

```
    protected AbstractNioByteChannel(Channel parent, EventLoop eventLoop,
SelectableChannel ch) {
        super(parent, eventLoop, ch, SelectionKey.OP_READ);
    }
```

到此，新接入的客户端连接处理完成，可以进行网络读写等 I/O 操作。

13.4　总结

本章首先对原生 NIO 类库的使用复杂性进行了讲解，然后对 Netty 服务端创建的时序图和步骤进行了详细地说明，随后结合 Netty 的源码对服务端创建进行剖析，最后对新的客户端的接入进行了源码层面的分析和讲解。

通过本章的学习，希望广大读者能够掌握 Netty 服务端创建的要点，并能够在实际的工作中正确的使用服务端相关的类库，编写出高效的业务代码。