



链滴

Netty|01 入门学习

作者: [qq692310342](#)

原文链接: <https://ld246.com/article/1566020472457>

来源网站: [链滴](#)

许可协议: [署名-相同方式共享 4.0 国际 \(CC BY-SA 4.0\)](#)



1.概述

- 1、Netty 是由 JBOSS 提供的一个 Java 开源框架。Netty 提供异步的、基于事件驱动的网络 应用程序框架，用以快速开发高性能、高可靠的网络 IO 程序。
- 2、Netty 是一个基于 NIO 的网络编程框架，使用 Netty 可以帮助你快速、简单的开发出一个网络用，相当于简化和流程化了 NIO 的开发过程。
- 3、作为当前最流行的 NIO 框架，Netty 在互联网领域、大数据分布式计算领域、游戏行业、通信行等获得了广泛的应用，知名的 Elasticsearch、Dubbo 框架内部都采用了 Netty。

2.整体设计

2.1 线程模型

1. 单线程模型：

服务器端用一个线程通过多路复用搞定所有的 IO 操作（包括连接，读、写等），编码简单，清晰明，但是如果客户端连接数量较多，将无法支撑，NIO编程技术就是典型的单线程模型！

2. 线程池模型

服务器端采用一个线程专门处理客户端连接请求，采用一个线程池负责 IO 操作。在绝大多数场景下该模型都能满足使用。

3. Netty模型

比较类似于上面的线程池模型，Netty 抽象出两组线程池，BossGroup 专门负责接收客户端连接，WorkerGroup 专门负责网络读写操作。NioEventLoop 表示一个不断循环执行处理 任务的线程，每个 ioEventLoop 都有一个 selector，用于监听绑定在其上的 socket 网络通道。NioEventLoop 内部采串行化设计，从消息的读取->解码->处理->编码->发送，始终由 IO 线程 NioEventLoop 负责。

一个 NioEventLoopGroup 下包含多个 NioEventLoop

每个 NioEventLoop 中包含有一个 Selector，一个 taskQueue

每个 NioEventLoop 的 Selector 上可以注册监听多个 NioChannel

每个 NioChannel 只会绑定在唯一的 NioEventLoop 上

每个 NioChannel 都绑定有一个自己的 ChannelPipeline

2.2 异步模型

1. Future和CallBack

Netty 的异步模型是建立在 future 和 callback 的之上的。callback 大家都比较熟悉了，这里重点说 Future，它的核心思想是：假设一个方法 fun，计算过程可能非常耗时，等待 fun 返回显然不合适那么可以在调用 fun 的时候，立马返回一个 Future，后续可以通过 Future 去监控方法 fun 的处理程。

在使用 Netty 进行编程时，拦截操作和转换出入站数据只需要您提供 callback 或利用 future 即可。使得链式操作简单、高效，并有利于编写可重用的、通用的代码。Netty 框架的目标就是让你的业务逻辑从网络基础应用编码中分离出来、解脱出来。

2. Handler

首先需要确立的一个观念就是，Netty 是一个基于链式的开发的模式，这里可以理解为我们可以自定许多的Handler来放在这个处理消息的链子（pipeline）上，而这些Handler就是我们在实际的开发需要编写的。

3. 核心API

3.1 ChannelHandler以及其实现类

大体分为：channelHandlerAdapter、channelOutboundHandler和channelInboundHandler

源生ChannelHandler

接口只定义了这几个通用的方法，其他交给实现类来拓展

方法的何时被调用我写在了注释里。

```
public interface ChannelHandler {
```

```
    /**
     * Gets called after the {@link ChannelHandler} was added to the actual context and it's ready to handle events.
     */
    // 处理程序添加到管道时调用handlerAdded(...)
    void handlerAdded(ChannelHandlerContext ctx) throws Exception;

    /**
     * Gets called after the {@link ChannelHandler} was removed from the actual context and it doesn't handle events anymore.
     */
    // 处理程序删除管道时调用的方法
    void handlerRemoved(ChannelHandlerContext ctx) throws Exception;
```

```
/*
 * Gets called if a {@link Throwable} was thrown.
 *
 * @deprecated is part of {@link ChannelInboundHandler}
 */
@Deprecated
// 处理程序再出现异常的时候调用的方法
void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception;
```

ChannelInboundHandler实现：

再来看看我们最常用到的实现类ChannelInboundHandler的内部方法,方法使用说明我写在了内部注中

```
public interface ChannelInboundHandler extends ChannelHandler {

    /**
     * The {@link Channel} of the {@link ChannelHandlerContext} was registered with its {@link EventLoop}
     */
    // 管道被注册进netty的时候调用，只会调用一次
    void channelRegistered(ChannelHandlerContext ctx) throws Exception;

    /**
     * The {@link Channel} of the {@link ChannelHandlerContext} was unregistered from its {@link EventLoop}
     */
    // 在通道没有被注册进来的时候调用方法
    void channelUnregistered(ChannelHandlerContext ctx) throws Exception;

    /**
     * The {@link Channel} of the {@link ChannelHandlerContext} is now active
     */
    // 通道触发的时候触发的方法
    void channelActive(ChannelHandlerContext ctx) throws Exception;

    /**
     * The {@link Channel} of the {@link ChannelHandlerContext} was registered is now inactive
     * and reached its
     * end of lifetime.
     */
    // 通道未就绪但是还在注册中触发的方法
    void channelInactive(ChannelHandlerContext ctx) throws Exception;

    /**
     * Invoked when the current {@link Channel} has read a message from the peer.
     */
    // 通道在识别到新消息的时候触发的事件
    void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception;

    /**
     * Invoked when the last message read by the current read operation has been consumed
     */
}
```

```

    * {@link #channelRead(ChannelHandlerContext, Object)}. If {@link ChannelOption#AUTO
READ} is off, no further
    * attempt to read an inbound data from the current {@link Channel} will be made until
    * {@link ChannelHandlerContext#read()} is called.
    */
// 通道读取一次数据完毕后触发的事件
void channelReadComplete(ChannelHandlerContext ctx) throws Exception;

/**
 * Gets called if an user event was triggered.
 */
// 在触发用户事件时调用
void userEventTriggered(ChannelHandlerContext ctx, Object evt) throws Exception;

/**
 * Gets called once the writable state of a {@link Channel} changed. You can check the state
with
 * {@link Channel#isWritable()}.
 */
// 在可写状态被更改的时候 调用方法
void channelWritabilityChanged(ChannelHandlerContext ctx) throws Exception;

/**
 * Gets called if a {@link Throwable} was thrown.
 */
@Override
@SuppressWarnings("deprecation")
// 通道出现异常触发的事件
void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception;
}

```

SimpleChannelInboundHandler接口的实现：

```

public abstract class SimpleChannelInboundHandler<I> extends ChannelInboundHandlerAda
ter {

    private final TypeParameterMatcher matcher;
    private final boolean autoRelease;

    /**
     * see {@link #SimpleChannelInboundHandler(boolean)} with {@code true} as boolean par
meter.
     */
    protected SimpleChannelInboundHandler() {
        this(true);
    }
    @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) {
        // 释放资源
        ReferenceCountUtil.release(msg);
    }
}
}

```

ChannelInboundHandlerAdapter接口实现：

```
public class ChannelInboundHandlerAdapter extends ChannelHandlerAdapter implements ChannelInboundHandler {
```

ChannelInboundHandlerAdapter的每个方法的默认实现都是通过ChannelHandlerContext将IO事或接收到的数据，传给所在的ChannelPipeline的下一个ChannelInboundHandler:

需要注意的几个点：

1、业务处理逻辑处理：用户可以通过拓展ChannelInboundHandlerAdapter，重写相应的方法来，成新的子类的方式来定义业务需要的处理逻辑，Netty默认针对特定功能的处理，提供了一些 ChannelInboundHandler的实现类

对于从Channel读入的数据，在调用channelRead方法处理时，默认实现也是传给下一个ChannelInboundHandler处理，不会销毁该数据对象，释放掉该数据所占用的空间的，如果不需要继续往下传输则可以调用ReferenceCountUtil.release(msg)手动释放掉。

3、但是对于实现类SimpleChannelInboundHandler来说，重写了ChannelInboundHandler的channelRead方法，默认会在最后释放掉msg的资源，所以是无法保存msg的引用的。

而SimpleChannelInboundHandler还新增了一个方法channelRead0（在netty 5.0之后channelRead0方法名称变成了messageReceived），这个方法的特殊在于，它是只会在msg通过了编码器解码器后才会执行的方法，参数本身会帮你转换为类上写的泛型，而这个泛型的数据类型就是在pipeline链增加的编码解码器中对应的类型。

值得注意的是，如果没有进行对应的编码解码就直接重写channelRead0方法的话，netty既不会处理来的msg，也不会报错异常，这是一个很坑的点。

数据的保留问题：如果用户在实现channelRead0方法自定义数据处理逻辑时，需要将该数据传给下一个ChannelInboundHandler，则需要调用ReferenceCountUtil.retain(msg)方法，原理是将msg的引用计数加1，因为ReferenceCountUtil.release(msg)是将msg的引用计数减1，同时当引用计数变成0时，释放该数据：（参考StringHandler）

```
public class StringHandler extends SimpleChannelInboundHandler<String> {

    @Override
    protected void channelRead0(ChannelHandlerContext ctx, String message)
        throws Exception {
        System.out.println(message);
        // 不释放数据，交给下一个ChannelInboundHandler继续处理
        ReferenceCountUtil.retain(message);
        ctx.fireChannelRead(message);
    }
}
```

```
}
```

ChannelOutboundHandler的接口实现：

```
public interface ChannelOutboundHandler extends ChannelHandler {  
    /**  
     * Called once a bind operation is made.  
     *  
     * @param ctx      the {@link ChannelHandlerContext} for which the bind operation is made  
     * @param localAddress  the {@link SocketAddress} to which it should bind  
     * @param promise    the {@link ChannelPromise} to notify once the operation completes  
     * @throws Exception thrown if an error occurs  
     */  
    void bind(ChannelHandlerContext ctx, SocketAddress localAddress, ChannelPromise promise)  
        throws Exception;  
  
    /**  
     * Called once a connect operation is made.  
     *  
     * @param ctx      the {@link ChannelHandlerContext} for which the connect operation is made  
     * @param remoteAddress  the {@link SocketAddress} to which it should connect  
     * @param localAddress  the {@link SocketAddress} which is used as source on connect  
     * @param promise    the {@link ChannelPromise} to notify once the operation completes  
     * @throws Exception thrown if an error occurs  
     */  
    void connect(  
        ChannelHandlerContext ctx, SocketAddress remoteAddress,  
        SocketAddress localAddress, ChannelPromise promise) throws Exception;  
  
    /**  
     * Called once a disconnect operation is made.  
     *  
     * @param ctx      the {@link ChannelHandlerContext} for which the disconnect operation is made  
     * @param promise    the {@link ChannelPromise} to notify once the operation completes  
     * @throws Exception thrown if an error occurs  
     */  
    void disconnect(ChannelHandlerContext ctx, ChannelPromise promise) throws Exception;  
  
    /**  
     * Called once a close operation is made.  
     *  
     * @param ctx      the {@link ChannelHandlerContext} for which the close operation is made  
     * @param promise    the {@link ChannelPromise} to notify once the operation completes  
     * @throws Exception thrown if an error occurs  
     */  
    void close(ChannelHandlerContext ctx, ChannelPromise promise) throws Exception;
```

```

/**
 * Called once a deregister operation is made from the current registered {@link EventLoop}
 *
 * @param ctx      the {@link ChannelHandlerContext} for which the close operation is
 * made
 * @param promise  the {@link ChannelPromise} to notify once the operation comple-
 * es
 * @throws Exception thrown if an error occurs
 */
void deregister(ChannelHandlerContext ctx, ChannelPromise promise) throws Exception;

/**
 * Intercepts {@link ChannelHandlerContext#read()}.
 */
void read(ChannelHandlerContext ctx) throws Exception;

/**
 * Called once a write operation is made. The write operation will write the messages throu-
 * h the
 * {@link ChannelPipeline}. Those are then ready to be flushed to the actual {@link Channel}
 * once
 * {@link Channel#flush()} is called
 *
 * @param ctx      the {@link ChannelHandlerContext} for which the write operation is
 * made
 * @param msg      the message to write
 * @param promise  the {@link ChannelPromise} to notify once the operation comple-
 * es
 * @throws Exception thrown if an error occurs
 */
void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise) throws Excep-
ion;

/**
 * Called once a flush operation is made. The flush operation will try to flush out all previou-
written messages
 * that are pending.
 *
 * @param ctx      the {@link ChannelHandlerContext} for which the flush operation is
 * made
 * @throws Exception thrown if an error occurs
 */
void flush(ChannelHandlerContext ctx) throws Exception;
}

```

对于ChannelOutboundHandler需要注意的点：

- 与ChannelInboundHandler类似，方法默认实现也是通过ChannelHandlerContext将写出的数据，交给下一个ChannelOutboundHandler处理。

ChannelOutboundHandlerAdapter接口实现：

```
public class ChannelOutboundHandlerAdapter extends ChannelHandlerAdapter implements  
hannelOutboundHandler {
```

可以看出：与ChannelInboundHandlerAdapter的作用类似，方法默认实现也是通过ChannelHandlerContext将写出的数据，交给下一个ChannelOutboundHandler处理。

ChannelDuplexHandler的接口实现：

```
public class ChannelDuplexHandler extends ChannelInboundHandlerAdapter implements Ch  
nnelOutboundHandler {
```

由此可知：ChannelDuplexHandler具备ChannelInboundHandler和ChannelInboundHandler的能力，可以实现读与写的功能，处理完数据后也是传递给下一个ChannelDuplexHandler处理

几个疑惑的点：

1、ChannelInboundHandler和ChannelOutboundHandler的关系：

ChannelInboundHandler用于定义对读入IO事件的处理，而ChannelOutboundHandler用于定义出IO事件的处理。

2、对于ChannelHandler里面的方法的执行顺序：将会在下一篇博客中单独做出解释

3、Netty在handler子模块，针对不同的功能，包括流量控制，数据flush，ip过滤，日志，ssl，大数流处理，超时心跳检测，拥塞控制，提供了相应的ChannelHandler实现类。 ---引用

3.2 ChannelHandlerContext 上下文API

这是事件处理器上下文对象，Pipeline 链中的实际处理节点。每个处理节点 ChannelHandlerContext 中包含一个具体的事件处理器 ChannelHandler，同时 ChannelHandlerContext 中也绑定对应的pipeline和Channel的信息，方便对ChannelHandler 进行调用。常用方法如下所示：

- ChannelFutureclose()，关闭通道
- ChannelOutboundInvokerflush()，刷新
- ChannelFuture writeAndFlush(Object msg)，将数据写到 ChannelPipeline 中当前 ChannelHandler 的下一个 ChannelHandler 开始处理（出站）

3.3 ChannelOption的API

Netty 在创建 Channel 实例后，一般都需要设置 ChannelOption 参数

ChannelOption 是 Socket 的标准参数，而非 Netty 独创的。常用的参数

1. ChannelOption.SO_BACKLOG 对应 TCP/IP 协议 listen 函数中的 backlog 参数，用来初始化服务器可连接队列大小。服务端处理客户端连接请求是顺序处理的，所以同一时间只能处理一个客户端连接。多个客户端来的时候，服务端将不能处理的客户端连接请求放在队列中等待处理，backlog 参数定了队列的大小。
2. ChannelOption.SO_KEEPALIVE，一直保持连接活动状态。

3.4 ChannelFuture 的API

表示 Channel 中异步 I/O 操作的结果，在 Netty 中所有的 I/O 操作都是异步的，I/O 的调用会直接回，调用者并不能立刻获得结果，但是可以通过 ChannelFuture 来获取 I/O 操作的处理状态。

常用方法如下所示：

- 1.Channelchannel()，返回当前正在进行 IO 操作的通道
- 2.ChannelFuturesync()，等待异步操作执行完毕

EventLoopGroup 和其实现类 NioEventLoopGroup

EventLoopGroup 是一组 EventLoop 的抽象，Netty 为了更好的利用多核 CPU 资源，一般会有多个 EventLoop 同时工作，每个 EventLoop 维护着一个 Selector 实例。

EventLoopGroup 提供 next 接口，可以从组里面按照一定规则获取其中一个 EventLoop 来处理任务。在 Netty 服务器端编程中，我们一般都需要提供两个 EventLoopGroup，如：BossEventLoopGroup 和 WorkerEventLoopGroup。

通常一个服务端口即一个ServerSocketChannel对应一个Selector和一个EventLoop线程。BossEventLoop 负责接收客户端的连接并将 SocketChannel 交给 WorkerEventLoopGroup 来进行 IO 处理

BossEventLoopGroup 通常是一个单线程的 EventLoop，EventLoop 维护着一个注册了 ServerSocketChannel 的 Selector 实例，BossEventLoop 不断轮询 Selector 将连接事件分离出来，通常是 OP_ACCEPT 事件，然后将接收到的 SocketChannel 交给 WorkerEventLoopGroup，WorkerEventLoopGroup 会由 next 选择其中一个 EventLoopGroup 来将这个 SocketChannel 注册到其维护的 Selector 对其后续的 IO 事件进行处理。常用方法如下所示：

- 1.publicNioEventLoopGroup()，构造方法
- 2.publicFuture<?>shutdownGracefully()，断开连接，关闭线程

3.5 ServerBootstrap 和 Bootstrap 的 API

ServerBootstrap 是 Netty 中的服务器端启动助手，通过它可以完成服务器端的各种配置；Bootstrap 是 Netty 中的客户端启动助手，通过它可以完成客户端的各种配置。常用方法如下所示：

- 1.publicServerBootstrapgroup(EventLoopGroup parentGroup,EventLoopGroupchildGroup)，该方法用于服务器端，用来设置两个 EventLoop
- 2.publicBgroup(EventLoopGroupgroup)，该方法用于客户端，用来设置一个 EventLoop
- 3.publicBchannel(Class<?extendsC>channelClass)，该方法用来设置一个服务器端的通道实现
- 4.public<T>Boption(ChannelOption<T>option,Tvalue)，用来给 ServerChannel 添加配置
- 5.public<T>ServerBootstrapchildOption(ChannelOption<T>childOption,Tvalue)，用来给接收的通道添加配置
- 6.public ServerBootstrapchildHandler(ChannelHandler childHandler)，该方法用来设置业务处理类（自定义的 handler）
- 7.publicChannelFuturebind(intinetPort)，该方法用于服务器端，用来设置占用的端口号
- 8.publicChannelFutureconnect(StringinetHost,intinetPort)，该方法用于客户端，用来连接服务端

3.6 Unpooled 的 API

这是 Netty 提供的一个专门用来操作缓冲区的工具类，常用方法如下

publicstaticByteBufcopiedBuffer(CharSequencestring,Charsetcharset)，通过给定的数据和字符

码返回一个 ByteBuf 对象 (类似于 NIO 中的 ByteBuffer 对象)

4. 编码和解码

4.1 概述

我们在编写网络应用程序的时候需要注意codec(编解码器)，因为数据在网络中传输的都是二进制字码数据，而我们拿到的目标数据往往不是字节码数据。因此在发送数据时就需要编码，接收数据时就要解码。

codec 的组成部分有两个：decoder(解码器)和 encoder(编码器)。

encoder 负责把业务数据转换成字节码数据，decoder 负责把字节码数据转换成业务数据。

Java 的序列化技术就可以作为 codec 去使用，

但是它的硬伤太多：

1. 无法跨语言，这应该是 Java 序列化最致命的问题了。
2. 序列化后的体积太大，是二进制编码的 5 倍多。
3. 序列化性能太低。

4.2 Netty自带的编解码器

1. StringEncoder，对字符串数据进行编码
2. ObjectEncoder，对 Java 对象进行编码

Netty 本身自带的 ObjectDecoder 和 ObjectEncoder 可以用来实现 POJO 对象或各种业务对象的码和解码，但其内部使用的仍是 Java 序列化技术，所以我们不建议使用。因此对于 POJO 对象或各业务对象要实现编码和解码，我们需要更高效更强的技术，这时就需要使用一些插件了。推荐使用 Google 的 Protobuf

END

2019年8月9日16:57:47