crazyhzm 265

Dubbo源码解析（十七）远程通信——Netty4

java dubbo netty 阅读约 27 分钟

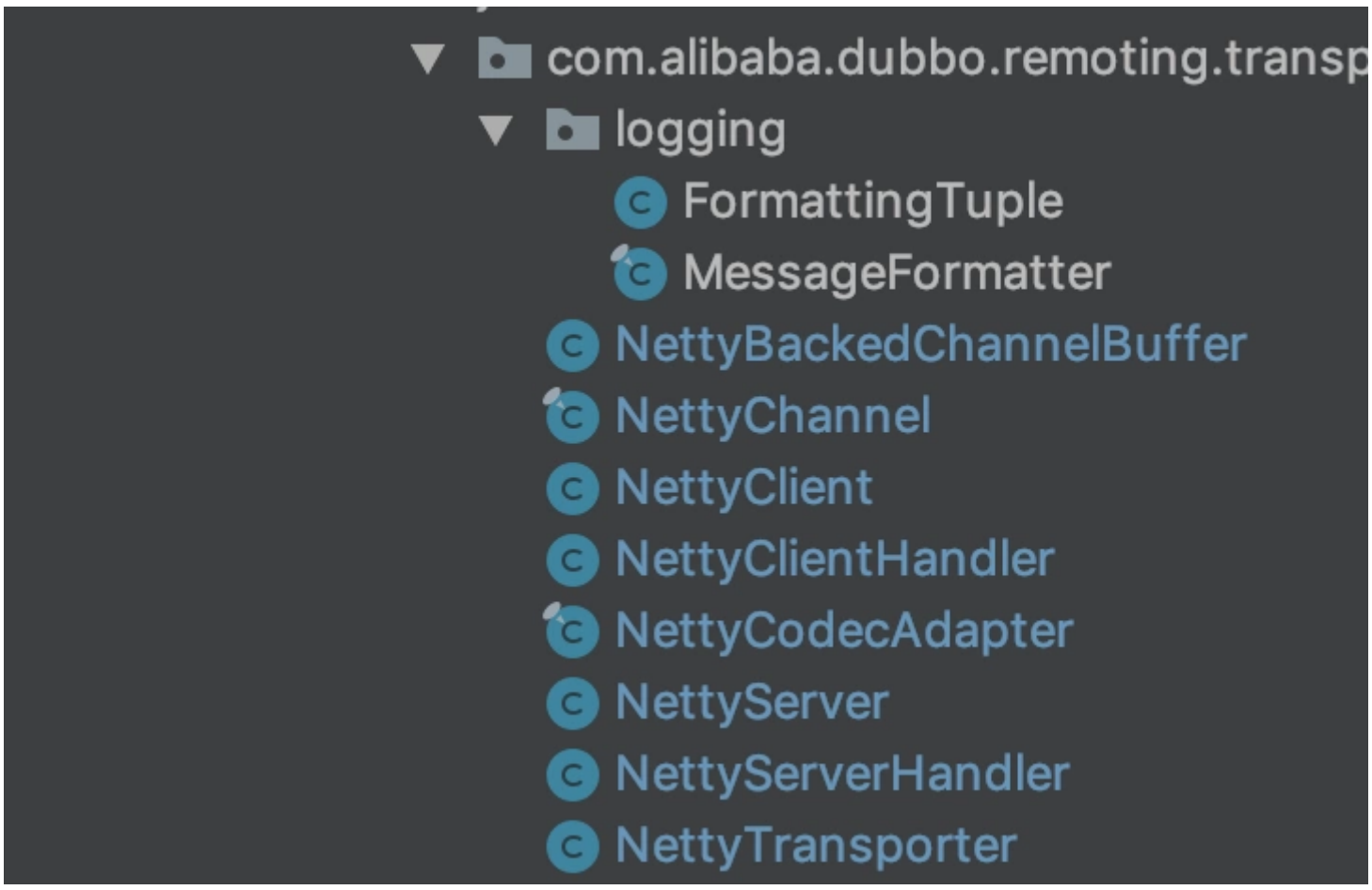
远程通讯——Netty4

> 目标：介绍基于netty4的来实现的远程通信、介绍dubbo-remoting-netty4内的源码解析。

前言

netty4对netty3兼容性不是很好，并且netty4在很多的术语和api也发生了改变，导致升级netty4会很艰辛，网上应该有很多相关文章，高版本的总有高版本的优势所在，所以dubbo也需要与时俱进，又新增了基于netty4来实现远程通讯模块。下面讲解的，如果跟上一篇文章有重复的地方我就略过去了。关键还是要把远程通讯的api那几篇看懂，看这几篇实现才会很简单。

下面是包的结构：



源码分析

（一）NettyChannel

该类继承了AbstractChannel，是基于netty4的通道实现类

1.属性

```
/**
 * 通道集合
 */
private static final ConcurrentMap<Channel, NettyChannel> channelMap = new ConcurrentHashMap<Channel, NettyChannel>();

/**
 * 通道
 */
private final Channel channel;

/**
 * 属性集合
 */
private final Map<String, Object> attributes = new ConcurrentHashMap<String, Object>();
```

属性跟netty3实现的通道类属性几乎一样，我就不讲解了。

2.getOrAddChannel

```
static NettyChannel getOrAddChannel(Channel ch, URL url, ChannelHandler handler) {
    if (ch == null) {
        return null;
    }
    // 首先从集合中取通道
    NettyChannel ret = channelMap.get(ch);
    // 如果为空，则新建
    if (ret == null) {
        NettyChannel nettyChannel = new NettyChannel(ch, url, handler);
        // 如果通道还活跃着
        if (ch.isActive()) {
            // 加入集合
            ret = channelMap.putIfAbsent(ch, nettyChannel);
        }
        if (ret == null) {
            ret = nettyChannel;
        }
    }
    return ret;
}
```

该方法是获得通道，如果集合中没有找到对应通道，则创建一个，然后加入集合。

3.send

```
@Override
public void send(Object message, boolean sent) throws RemotingException {
    super.send(message, sent);

    boolean success = true;
    int timeout = 0;
    try {
        // 写入数据，发送消息
        ChannelFuture future = channel.writeAndFlush(message);
        // 如果已经发送过
        if (sent) {
            // 获得超时时间
            timeout = getUrl().getPositiveParameter(Constants.TIMEOUT_KEY, Constants.DEFAULT_TIMEOUT);
            // 等待timeout的连接时间后查看是否发送成功
            success = future.await(timeout);
        }
        // 获得异常
        Throwable cause = future.cause();
        // 如果异常不为空，则抛出异常
        if (cause != null) {
            throw cause;
        }
    } catch (Throwable e) {
        throw new RemotingException(this, "Failed to send message " + message + " to " + getRemoteAddress() + ",
cause: " + e.getMessage(), e);
    }
```

该方法是发送消息，调用了channel.writeAndFlush方法，与netty3的实现只是调用的api不同。

4.close

```
@Override
public void close() {
    try {
        super.close();
    } catch (Exception e) {
        logger.warn(e.getMessage(), e);
    }
    try {
        // 移除通道
        removeChannelIfDisconnected(channel);
    } catch (Exception e) {
        logger.warn(e.getMessage(), e);
    }
    try {
        // 清理属性集合
        attributes.clear();
    } catch (Exception e) {
        logger.warn(e.getMessage(), e);
    }
    try {
        if (logger.isInfoEnabled()) {
            logger.info("Close netty channel " + channel);
        }
        // 关闭通道
        channel.close();
    } catch (Exception e) {
```

该方法就是操作了四个步骤，比较清晰。

（二）NettyClientHandler

该类继承了ChannelDuplexHandler，是基于netty4实现的客户端通道处理实现类。这里的设计与netty3实现的通道处理器有所不同，netty3实现的通道处理器是被客户端和服务端统一使用的，而在这里服务端和客户端使用了两个不同的Handler来处理。并且netty3的NettyHandler是基于netty3的SimpleChannelHandler设计的，而这里是基于netty4的ChannelDuplexHandler。

```
/**
 * url对象
 */
private final URL url;

/**
 * 通道
 */
private final ChannelHandler handler;
```

该类的属性只有两个，下面实现的方法也都是调用了handler的方法，我就举一个例子：

```
@Override
public void disconnect(ChannelHandlerContext ctx, ChannelPromise future)
        throws Exception {
    // 获得通道
    NettyChannel channel = NettyChannel.getOrAddChannel(ctx.channel(), url, handler);
    try {
        // 断开连接
        handler.disconnected(channel);
    } finally {
        // 从集合中移除
        NettyChannel.removeChannelIfDisconnected(ctx.channel());
    }
}
```

可以看到分了三部，获得通道对象，调用handler方法，最后检测一下通道是否活跃。其他方法也是差不多。

（三）NettyServerHandler

该类继承了ChannelDuplexHandler，是基于netty4实现的服务端通道处理实现类。

```
/**
 * 连接该服务器的通道数 key为ip:port
 */
private final Map<String, Channel> channels = new ConcurrentHashMap<String, Channel>(); // <ip:port, channel>

/**
 * url对象
 */
private final URL url;

/**
 * 通道处理器
 */
private final ChannelHandler handler;
```

该类有三个属性，比NettyClientHandler多了一个属性channels，下面的实现方法也是一样的，都是调用了handler方法，来看一个例子：

```
@Override
public void channelActive(ChannelHandlerContext ctx) throws Exception {
    // 激活事件
    ctx.fireChannelActive();

    // 获得通道
    NettyChannel channel = NettyChannel.getOrAddChannel(ctx.channel(), url, handler);
    try {
        // 如果通道不为空，则加入集合中
        if (channel != null) {
            channels.put(NetUtils.toAddressString((InetSocketAddress) ctx.channel().remoteAddress()), channel);
        }
        // 连接该通道
        handler.connected(channel);
    } finally {
        // 如果通道不活跃，则移除通道
        NettyChannel.removeChannelIfDisconnected(ctx.channel());
    }
}
```

该方法是通道活跃的时候调用了handler.connected，差不多也是常规套路，就多了激活事件和加入到通道中。其他方法也差不多。

（四）NettyClient

该类继承了AbstractClient，是基于netty4实现的客户端实现类。

1.属性

```
/**
 * NioEventLoopGroup对象
 */
private static final NioEventLoopGroup nioEventLoopGroup = new NioEventLoopGroup(Constants.DEFAULT_IO_THREADS, new
DefaultThreadFactory("NettyClientWorker", true));

/**
 * 客户端引导类
 */
private Bootstrap bootstrap;

/**
 * 通道
 */
private volatile Channel channel; // volatile, please copy reference to use
```

属性里的NioEventLoopGroup对象是netty4中的对象，什么用处请看netty的解析。

2.doOpen

```
@Override
protected void doOpen() throws Throwable {
    // 创建一个客户端的通道处理器
    final NettyClientHandler nettyClientHandler = new NettyClientHandler(getUrl(), this);
    // 创建一个引导类
    bootstrap = new Bootstrap();
    // 设置可选项
    bootstrap.group(nioEventLoopGroup)
            .option(ChannelOption.SO_KEEPALIVE, true)
            .option(ChannelOption.TCP_NODELAY, true)
            .option(ChannelOption.ALLOCATOR, PooledByteBufAllocator.DEFAULT)
            //.option(ChannelOption.CONNECT_TIMEOUT_MILLIS, getTimeout())
            .channel(NioSocketChannel.class);

    // 如果连接超时时间小于3s，则设置为3s，也就是说最低的超时时间为3s
    if (getConnectTimeout() < 3000) {
        bootstrap.option(ChannelOption.CONNECT_TIMEOUT_MILLIS, 3000);
    } else {
        bootstrap.option(ChannelOption.CONNECT_TIMEOUT_MILLIS, getConnectTimeout());
    }

    // 创建一个客户端
    bootstrap.handler(new ChannelInitializer() {

        @Override
        protected void initChannel(Channel ch) throws Exception {
```

该方法还是做了创建客户端，并且打开的操作，其中很多的参数设置操作。

其他方法跟 dubbo源码解析（十六）远程通信——Netty3中写到的NettyClient实现一样。

（五）NettyServer

该类继承了AbstractServer，实现了Server。是基于netty4实现的服务器类

```
private static final Logger logger = LoggerFactory.getLogger(NettyServer.class);


/**
 * 连接该服务器的通道集合 key为ip:port
 */
private Map<String, Channel> channels; // <ip:port, channel>

/**
 * 服务器引导类
 */
private ServerBootstrap bootstrap;

/**
 * 通道
 */
private io.netty.channel.Channel channel;

/**
 * boss线程组
 */
private EventLoopGroup bossGroup;
/**
 * worker线程组
 */
private EventLoopGroup workerGroup;
```

属性相较netty3而言，新增了两个线程组，同样也是因为netty3和netty4的设计不同。

2.doOpen

```
@Override
protected void doOpen() throws Throwable {
    // 创建服务引导类
    bootstrap = new ServerBootstrap();

    // 创建boss线程组
    bossGroup = new NioEventLoopGroup(1, new DefaultThreadFactory("NettyServerBoss", true));
    // 创建worker线程组
    workerGroup = new NioEventLoopGroup(getUrl().getPositiveParameter(Constants.IO_THREADS_KEY,
Constants.DEFAULT_IO_THREADS),
            new DefaultThreadFactory("NettyServerWorker", true));

    // 创建服务器处理器
    final NettyServerHandler nettyServerHandler = new NettyServerHandler(getUrl(), this);
    // 获得通道集合
    channels = nettyServerHandler.getChannels();

    // 设置ventLoopGroup还有可选项
    bootstrap.group(bossGroup, workerGroup)
            .channel(NioServerSocketChannel.class)
            .childOption(ChannelOption.TCP_NODELAY, Boolean.TRUE)
            .childOption(ChannelOption.SO_REUSEADDR, Boolean.TRUE)
            .childOption(ChannelOption.ALLOCATOR, PooledByteBufAllocator.DEFAULT)
            .childHandler(new ChannelInitializer<NioSocketChannel>() {
                @Override
                protected void initChannel(NioSocketChannel ch) throws Exception {
```

该方法是创建服务器，并且开启。如果熟悉netty4点朋友应该觉得还是很好理解的。其他方法跟《 dubbo源码解析（十六）远程通信——Netty3》中写到的NettyClient实现一样，处理close中要多关闭两个线程组

（六）NettyTransporter

该类跟《dubbo源码解析（十六）远程通信——Netty3》中的NettyTransporter一样的实现。

（七）NettyCodecAdapter

该类是基于netty4的编解码器。

1.属性

```
/**
 * 编码器
 */
private final ChannelHandler encoder = new InternalEncoder();

/**
 * 解码器
 */
private final ChannelHandler decoder = new InternalDecoder();

/**
 * 编解码器
 */
private final Codec2 codec;

/**
 * url对象
 */
private final URL url;

/**
 * 通道处理器
 */
private final com.alibaba.dubbo.remoting.ChannelHandler handler;
```

属性跟基于netty3实现的编解码一样。

2.InternalEncoder

```
private class InternalEncoder extends MessageToByteEncoder {

    @Override
    protected void encode(ChannelHandlerContext ctx, Object msg, ByteBuf out) throws Exception {
        // 创建缓冲区
        com.alibaba.dubbo.remoting.buffer.ChannelBuffer buffer = new NettyBackedChannelBuffer(out);
        // 获得通道
        Channel ch = ctx.channel();
        // 获得netty通道
        NettyChannel channel = NettyChannel.getOrAddChannel(ch, url, handler);
        try {
            // 编码
            codec.encode(channel, buffer, msg);
        } finally {
            // 检测通道是否活跃
            NettyChannel.removeChannelIfDisconnected(ch);
        }
    }
}
```

该内部类是编码器的抽象，主要的编码还是调用了codec.encode。

3.InternalDecoder

```
private class InternalDecoder extends ByteToMessageDecoder {

    @Override
    protected void decode(ChannelHandlerContext ctx, ByteBuf input, List<Object> out) throws Exception {

        // 创建缓冲区
        ChannelBuffer message = new NettyBackedChannelBuffer(input);

        // 获得通道
        NettyChannel channel = NettyChannel.getOrAddChannel(ctx.channel(), url, handler);

        Object msg;

        int saveReaderIndex;

        try {
            // decode object.
            do {
                // 记录读索引
                saveReaderIndex = message.readerIndex();
                try {
                    // 解码
                    msg = codec.decode(channel, message);
                } catch (IOException e) {
                    throw e;
                }
```

该内部类是解码器的抽象类，其中关键的是调用了codec.decode。

（八）NettyBackedChannelBuffer

该类是缓冲区类。

```
/**
 * 缓冲区
 */
private ByteBuf buffer;
```

其中的方法几乎都调用了该属性的方法。而ByteBuf是netty4中的字节数据的容器。

（九）FormattingTuple和MessageFormatter

这两个类是用于用于格式化的，是从netty4中复制出来的，其中并且稍微做了一下改动。我就不再讲解了。

后记

> 该部分相关的源码解析地址：https://github.com/CrazyHZM/i...

该文章讲解了基于netty4的来实现的远程通信、介绍dubbo-remoting-netty4内的源码解析，关键需要对netty4有所了解。下一篇我会讲解基于zookeeper实现远程通信部分。

crazyhzm
265

关注作者

0 条评论

得票 • 时间

撰写评论 ...

提交评论

推荐阅读

dubbo源码解析——概要篇

这次源码解析借鉴《肥朝》前辈的dubbo源码解析，进行源码学习。总结起来就是先总体,后局部.也就是先把需要注意的概念先抛...

• 阅读 631

【源码分析】AsyncTask源码分析

AsyncTask类在早期的开发中占的比重非常重，因为它可以方便的将耗时工作与UI线程连接在一起，随着android版本的更新、业务...

summerpxy • 阅读 6

dubbo源码解析（二）Dubbo扩展机制SPI

前一篇文章《dubbo源码解析（一）Hello,Dubbo》是对dubbo整个项目大体的介绍，而从这篇文章开始，我将会从源码来解读dub...

CrazyHzm • 阅读 355

dubbo负载均衡策略及对应源码分析

在集群负载均衡时，Dubbo提供了多种均衡策略，缺省为random随机调用。我们还可以扩展自己的负责均衡策略，前提是你已经...

bali • 阅读 947

Spring-boot+Dubbo应用启停源码分析

背景介绍DubboSpringBoot工程致力于简化DubboRPC框架在SpringBoot应用场景的开发。同时也整合了SpringBoot特性：你有没...

阿里云云栖社区 • 阅读 13

dubbo源码解析——消费过程