crazyhzm 265

Dubbo源码解析（十五）远程通信——Mina

java dubbo mina 阅读约 34 分钟

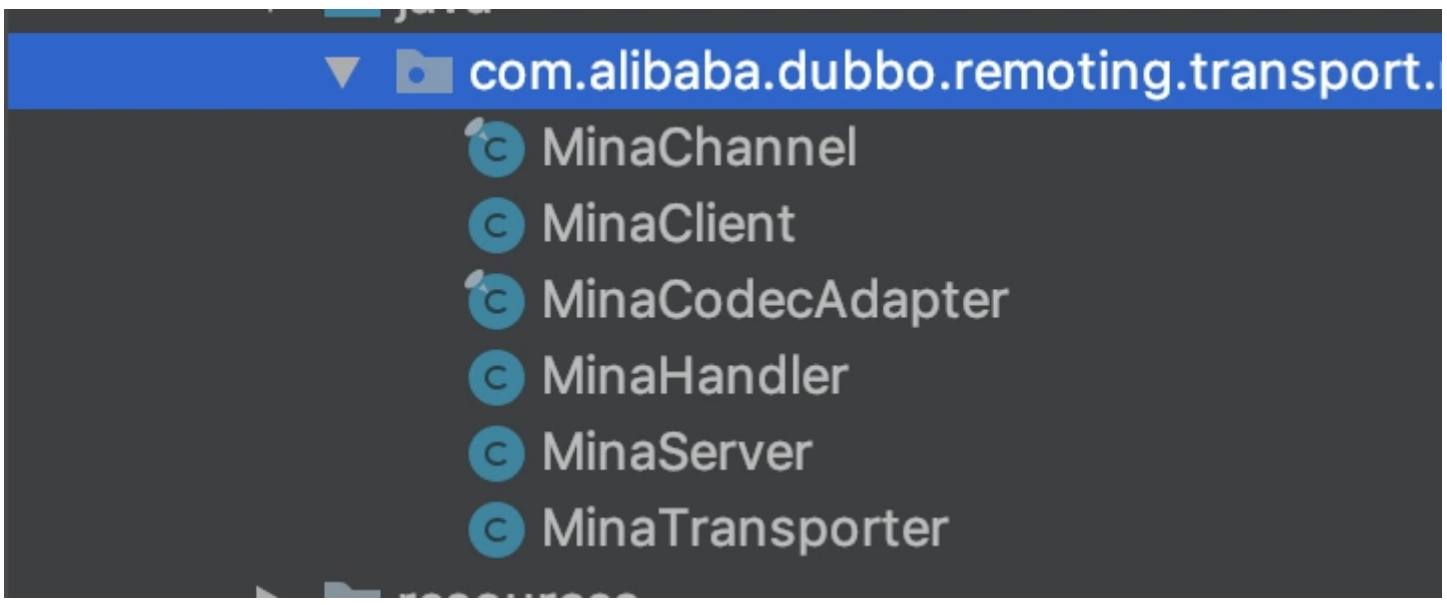
远程通讯——Mina

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

前言

Apache MINA是一个网络应用程序框架，可帮助用户轻松开发高性能和高可扩展性的网络应用程序。它通过Java NIO在各种传输（如TCP / IP和UDP / IP）上提供抽象的事件驱动异步API。它通常被称为NIO框架库、客户端服务器框架库或者网络套接字库。那么本问就要讲解在dubbo项目中，基于mina的API实现服务端和客户端来完成远程通讯这件事情。

下面是mina实现的包结构：



源码分析

（一）MinaChannel

该类继承了AbstractChannel，是基于mina实现的通道。

1.属性

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

/**
 * 通道的key
 */
private static final String CHANNEL_KEY = MinaChannel.class.getName() + ".CHANNEL";

/**
 * mina中的一个句柄，表示两个端点之间的连接，与传输类型无关
 */
private final IoSession session;
```

该类的属性除了封装了一个CHANNEL_KEY以外，还用到了mina中的IoSession，它封装着一个连接所需要的方法，比如获得远程地址等。

2.getOrAddChannel

```
static MinaChannel getOrAddChannel(IoSession session, URL url, ChannelHandler handler) {
    // 如果连接session为空，则返回空
    if (session == null) {
        return null;
    }
    // 获得MinaChannel实例
    MinaChannel ret = (MinaChannel) session.getAttribute(CHANNEL_KEY);
    // 如果不存在，则创建
    if (ret == null) {
        // 创建一个MinaChannel实例
        ret = new MinaChannel(session, url, handler);
        // 如果两个端点连接
        if (session.isConnected()) {
            // 把新创建的MinaChannel添加到session 中
            MinaChannel old = (MinaChannel) session.setAttribute(CHANNEL_KEY, ret);
            // 如果属性的旧值不为空，则重新设置旧值
            if (old != null) {
                session.setAttribute(CHANNEL_KEY, old);
                ret = old;
            }
        }
    }
    return ret;
}
```

该方法是一个获得MinaChannel对象的方法，其中每一个MinaChannel都会被放在session的属性值中。

3.removeChannelIfDisconnected

```
static void removeChannelIfDisconnected(IoSession session) {
    if (session != null && !session.isConnected()) {
        session.removeAttribute(CHANNEL_KEY);
    }
}
```

该方法是当没有连接时移除该通道，比较简单。

4.send

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

    boolean success = true;
    int timeout = 0;
    try {
        // 发送消息，返回future
        WriteFuture future = session.write(message);
        // 如果已经发送过了
        if (sent) {
            // 获得延迟时间
            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.join(timeout);
        }
    } catch (Throwable e) {
        throw new RemotingException(this, "Failed to send message " + message + " to " + getRemoteAddress() + ",
cause: " + e.getMessage(), e);
    }

    if (!success) {
        throw new RemotingException(this, "Failed to send message " + message + " to " + getRemoteAddress()
                + "in timeout(" + timeout + "ms) limit");
    }
}
```

该方法是最关键的发送消息，其中调用到了session的write方法，就是mina封装的发送消息。并且根据返回的WriteFuture对象来判断是否发送成功。

（二）MinaHandler

该类继承了IoHandlerAdapter，是通道处理器实现类，其中就是mina项目中IoHandler接口的几个 方法。

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

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

该类有两个属性，上述提到的实现IoHandler接口方法都是调用了handler来实现的，我就举例讲一个，其他的都一样的写法：

```
@Override
public void sessionOpened(IoSession session) throws Exception {
    // 获得MinaChannel对象
    MinaChannel channel = MinaChannel.getOrAddChannel(session, url, handler);
    try {
        // 调用接连该通道
        handler.connected(channel);
    } finally {
        // 如果没有连接则移除通道
        MinaChannel.removeChannelIfDisconnected(session);
    }
}
```

该方法在IoHandler中叫做sessionOpened，其实就是连接方法，所以调用的是handler.connected。其他方法也一样，请自行查看。

（三）MinaClient

该类继承了AbstractClient类，是基于mina实现的客户端类。

1.属性

```
/**
 * 套接字连接集合
 */
private static final Map<String, SocketConnector> connectors = new ConcurrentHashMap<String, SocketConnector>();

/**
 * 连接的key
 */
private String connectorKey;
/**
 * 套接字连接者
 */
private SocketConnector connector;

/**
 * 一个句柄
 */
private volatile IoSession session; // volatile, please copy reference to use
```

该类中的属性都跟mina项目中封装类有关系。

2.doOpen

```
@Override
protected void doOpen() throws Throwable {
    // 用url来作为key
    connectorKey = getUrl().toFullString();
    // 先从集合中取套接字连接
    SocketConnector c = connectors.get(connectorKey);
    if (c != null) {
        connector = c;
        // 如果为空
    } else {
        // set thread pool. 设置线程池
        connector = new SocketConnector(Constants.DEFAULT_IO_THREADS,
                Executors.newCachedThreadPool(new NamedThreadFactory("MinaClientWorker", true)));
        // config 获得套接字连接配置
        SocketConnectorConfig cfg = (SocketConnectorConfig) connector.getDefaultConfig();
        cfg.setThreadModel(ThreadModel.MANUAL);
        // 启用TCP_NODELAY
        cfg.getSessionConfig().setTcpNoDelay(true);
        // 启用SO_KEEPALIVE
        cfg.getSessionConfig().setKeepAlive(true);
        int timeout = getConnectTimeout();
        // 设置连接超时时间
        cfg.setConnectTimeout(timeout < 1000 ? 1 : timeout / 1000);
        // set codec.
        // 设置编解码器
        connector.getFilterChain().addLast("codec", new ProtocolCodecFilter(new MinaCodecAdapter(getCodec(),
```

该方法是打开客户端，在mina中用套接字连接者connector来表示。其中的操作就是新建一个connector，并且设置相应的属性，然后加入集合。

3.doConnect

```
@Override
protected void doConnect() throws Throwable {
    // 连接服务器
    ConnectFuture future = connector.connect(getConnectAddress(), new MinaHandler(getUrl(), this));
    long start = System.currentTimeMillis();
    final AtomicReference<Throwable> exception = new AtomicReference<Throwable>();
    // 用于对线程的阻塞和唤醒
    final CountDownLatch finish = new CountDownLatch(1); // resolve future.awaitUninterruptibly() dead lock
    // 加入监听器
    future.addListener(new IoFutureListener() {
        @Override
        public void operationComplete(IoFuture future) {
            try {
                // 如果已经读完了
                if (future.isReady()) {
                    // 创建获得该连接的IoSession实例
                    IoSession newSession = future.getSession();
                    try {
                        // Close old channel 关闭旧的session
                        IoSession oldSession = MinaClient.this.session; // copy reference
                        if (oldSession != null) {
                            try {
                                if (logger.isInfoEnabled()) {
                                    logger.info("Close old mina channel " + oldSession + " on create new mina
channel " + newSession);
                                }
```

该方法是客户端连接服务器的实现方法。其中用到了CountDownLatch来代表完成完成事件，它来做一个线程等待，直到1个线程完成上述的动作，也就是连接完成结束，才释放等待的线程。保证每次只有一条线程去连接，解决future.awaitUninterruptibly（）死锁问题。

其他方法请自行查看我写的注释。

（四）MinaServer

该类继承了AbstractServer，是基于mina实现的服务端实现类。

1.属性

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

/**
 * 套接字接收者对象
 */
private SocketAcceptor acceptor;
```

2.doOpen

```
@Override
protected void doOpen() throws Throwable {
    // set thread pool.
    // 创建套接字接收者对象
    acceptor = new SocketAcceptor(getUrl().getPositiveParameter(Constants.IO_THREADS_KEY,
Constants.DEFAULT_IO_THREADS),
            Executors.newCachedThreadPool(new NamedThreadFactory("MinaServerWorker",
                    true)));
    // config
    // 设置配置
    SocketAcceptorConfig cfg = (SocketAcceptorConfig) acceptor.getDefaultConfig();
    cfg.setThreadModel(ThreadModel.MANUAL);
    // set codec. 设置编解码器
    acceptor.getFilterChain().addLast("codec", new ProtocolCodecFilter(new MinaCodecAdapter(getCodec(), getUrl(),
this)));

    // 开启服务器
    acceptor.bind(getBindAddress(), new MinaHandler(getUrl(), this));
}
```

该方法是创建服务器，并且打开服务器。关键就是调用了acceptor的方法。

3.doClose

```
@Override
protected void doClose() throws Throwable {
    try {
        if (acceptor != null) {
            // 取消绑定，也就是关闭服务器
            acceptor.unbind(getBindAddress());
        }
    } catch (Throwable e) {
        logger.warn(e.getMessage(), e);
    }
}
```

该方法是关闭服务器，就是调用了acceptor.unbind方法。

4.getChannels

```
@Override
public Collection<Channel> getChannels() {
    // 获得连接到该服务器到所有连接句柄
    Set<IoSession> sessions = acceptor.getManagedSessions(getBindAddress());
    Collection<Channel> channels = new HashSet<Channel>();
    for (IoSession session : sessions) {
        if (session.isConnected()) {
            // 每次都用一个连接句柄创建一个通道
            channels.add(MinaChannel.getOrAddChannel(session, getUrl(), this));
        }
    }
    return channels;
}
```

该方法是获得所有连接该服务器的通道。

5.getChannel

```
@Override
public Channel getChannel(InetSocketAddress remoteAddress) {
    // 获得连接到该服务器到所有连接句柄
    Set<IoSession> sessions = acceptor.getManagedSessions(getBindAddress());
    // 遍历所有句柄，找到要找的通道
    for (IoSession session : sessions) {
        if (session.getRemoteAddress().equals(remoteAddress)) {
            return MinaChannel.getOrAddChannel(session, getUrl(), this);
        }
    }
    return null;
}
```

该方法是获得地址对应的单个通道。

（五）MinaTransporter

```
public class MinaTransporter implements Transporter {

    public static final String NAME = "mina";

    @Override
    public Server bind(URL url, ChannelHandler handler) throws RemotingException {
        // 创建MinaServer实例
        return new MinaServer(url, handler);
    }

    @Override
    public Client connect(URL url, ChannelHandler handler) throws RemotingException {
        // 创建MinaClient实例
        return new MinaClient(url, handler);
    }

}
```

该类实现了Transporter接口，是基于mina的传输层实现。可以看到，bind和connect方法分别就是创建了MinaServer和MinaClient实例。这里我建议查看一下《dubbo源码解析（九）远程通信——Transport层》。

（六）MinaCodecAdapter

该类是基于mina实现的编解码类，实现了ProtocolCodecFactory。

1.属性

```
/**
 * 编码对象
 */
private final ProtocolEncoder encoder = new InternalEncoder();

/**
 * 解码对象
 */
private final ProtocolDecoder decoder = new InternalDecoder();

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

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

/**
 * 通道处理器对象
 */
private final ChannelHandler handler;

/**
```

属性比较好理解，该编解码器用到了ProtocolEncoder和ProtocolDecoder，而InternalEncoder和InternalDecoder两个类是该类的内部类，它们实现了ProtocolEncoder和ProtocolDecoder，关键的编解码逻辑在这两个类中实现。

2.构造方法

```
public MinaCodecAdapter(Codec2 codec, URL url, ChannelHandler handler) {
    this.codec = codec;
    this.url = url;
    this.handler = handler;
    int b = url.getPositiveParameter(Constants.BUFFER_KEY, Constants.DEFAULT_BUFFER_SIZE);
    // 如果缓存区大小在16字节以内，则设置配置大小，如果不是，则设置8字节的缓冲区大小
    this.bufferSize = b >= Constants.MIN_BUFFER_SIZE && b <= Constants.MAX_BUFFER_SIZE ? b :
Constants.DEFAULT_BUFFER_SIZE;
}
```

3.InternalEncoder

```
private class InternalEncoder implements ProtocolEncoder {

    @Override
    public void dispose(IoSession session) throws Exception {
    }

    @Override
    public void encode(IoSession session, Object msg, ProtocolEncoderOutput out) throws Exception {
        // 动态分配一个1k的缓冲区
        ChannelBuffer buffer = ChannelBuffers.dynamicBuffer(1024);
        // 获得通道
        MinaChannel channel = MinaChannel.getOrAddChannel(session, url, handler);
        try {
            // 编码
            codec.encode(channel, buffer, msg);
        } finally {
            // 检测是否断开连接，如果断开，则移除
            MinaChannel.removeChannelIfDisconnected(session);
        }
        // 写数据到out中
        out.write(ByteBuffer.wrap(buffer.toByteBuffer()));
        out.flush();
    }
}
```

该内部类是编码类，其中的encode方法中写到了编码核心调用的是codec.encode。

4.InternalDecoder

```
private class InternalDecoder implements ProtocolDecoder {

    private ChannelBuffer buffer = ChannelBuffers.EMPTY_BUFFER;

    @Override
    public void decode(IoSession session, ByteBuffer in, ProtocolDecoderOutput out) throws Exception {
        int readable = in.limit();
        if (readable <= 0) return;

        ChannelBuffer frame;

        // 如果缓冲区还有可读字节数
        if (buffer.readable()) {
            // 如果缓冲区是DynamicChannelBuffer类型的
            if (buffer instanceof DynamicChannelBuffer) {
                // 往buffer中写入数据
                buffer.writeBytes(in.buf());
                frame = buffer;
            } else {
                // 缓冲区大小
                int size = buffer.readableBytes() + in.remaining();
                // 动态分配一个缓冲区
                frame = ChannelBuffers.dynamicBuffer(size > bufferSize ? size : bufferSize);
                // buffer的数据把写到frame
                frame.writeBytes(buffer, buffer.readableBytes());
                // 把流中的数据写到frame
```

该内部类是解码类，其中decode方法中关键的是调用了codec.decode，其余的操作是利用缓冲区对数据的冲刷流转。

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mina的来实现的远程通信、介绍dubbo-remoting-mina内的源码解析，关键需要对mina有所了解。下一篇我会讲解基于netty3实现远程通信部分。